



**A SEMANTIC INTERFACE TO SCENARIO COMPONENT
REUSE IN DOD SIMULATION SYSTEMS**

THESIS

Lawrence A. Breighner, Captain, USAF

AFIT/GCS/ENG/01M-01

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED.

20010706 165

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense or U.S. Government.

A SEMANTIC INTERFACE TO SCENARIO COMPONENT REUSE IN DOD SIMULATION SYSTEMS

THESIS

Presented to the faculty of the Graduate School of Engineering & Management

of the Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Lawrence A. Breighner, B.S.

Captain, USAF

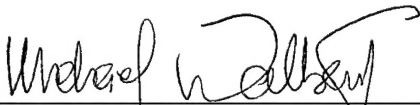
March 2001

Approved for public release, distribution unlimited.

A SEMANTIC INTERFACE TO SCENARIO COMPONENT REUSE IN DOD SIMULATION SYSTEMS

Lawrence A. Breighner, B.S.
Captain, USAF

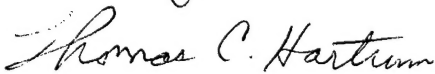
Approved:



Michael L. Talbert, Ph.D., Major, USAF
Chairman

20 Feb 2001

Date



Thomas C. Hartrum, Ph.D.
Committee member

20 Feb 2001

Date



Karl Mathias, Ph.D., Major, USAF
Committee member

20 Feb 2001

Date

ACKNOWLEDGMENTS

I owe much appreciation to those who have contributed to this work. To Major Michael Talbert, thanks for your encouragement and expert guidance, and for allowing me latitude in my research to explore and develop unfamiliar concepts and ideas. Thanks also to Major Karl Mathias for our discussions concerning the feasibility of a workable common object model, and to Dr. Tom Hartrum for his tireless efforts to bring this old student back up to speed.

Above all, my wife deserves much praise and admiration for her selflessness through our AFIT experience. She was always there to listen to my problems and grievances, and never complained about being an "AFIT widow." Even while pregnant with our son , she maintained her resilience and never grumbled about my absence at her maternity appointments.

Lawrence A. Breighner

TABLE OF CONTENTS

Table of Contents	v
Table of Figures	vii
Abstract	ix
A Semantic Interface to Scenario Component Reuse in DOD Simulation Systems	1
1. Introduction	1
1.1 Definition of Terms	2
1.2 Problem Statement	2
1.3 Research Focus	5
1.4 Summary	6
2. Literature Review	8
2.1 Introduction	8
2.2 Suppressor Overview	9
2.3 Multi-Spectral Force Deployment (MSFD)	13
2.4 CERTCORT Common Object Model	15
2.4.1 SUPPRESSOR Grammar Parser	15
2.4.2 SUPPRESSOR Object Model	17
2.4.3 Multi-Spectral Force Deployment (MSFD) Object Model	19
2.5 Metadata	21
2.5.1 RAND Metadata Management System	23
2.6 The Visitor Pattern	28
2.7 JTree	31
2.8 Agent Technology	32
2.8.1 Intelligent Agents	33
2.8.2 Agents and Objects	34
2.8.3 Agent Architectures	34
2.8.4 Multi-Agent Systems	36
2.9 Object-Oriented Database Management Systems	38
2.9.1 Object-Oriented Database System Manifesto	38
2.9.2 ObjectStore	39
2.10 Multi-Database Systems	40
2.11 Summary	42
3. Methodology	43
3.1 Introduction	43
3.2 Development Tools	43
3.2.1 Object Modeling	43
3.2.2 Programming Language	45
3.3 General Approach	46
3.3.1 Scenario Component Representation	46
3.3.2 Component Generation	49
3.3.3 Relevant Component Retrieval	53
3.3.4 Component Transformation	57

3.4	Semantic Broker Architecture	60
3.4.1	ScenarioRegistry Application	61
3.4.2	SemanticGateway Application	65
3.5	Summary	72
4.	Implementation	73
4.1	Introduction	73
4.2	Design Issues	73
4.2.1	Scenario Class Hierarchy.....	73
4.2.2	Component Generation.....	77
4.2.3	Signature Analysis and the SCDB.....	78
4.3	Semantic Broker Major Components	81
4.3.1	ScenarioRegistry Application	83
4.3.2	SemanticGateway Application	89
4.3.3	SemanticGateway – ScenarioRegistry Interaction	111
4.4	Semantic Broker Demonstration	112
4.4.1	Java Packages	113
4.4.2	Hardware and Software Platforms	113
4.4.3	Component Retrieval Test Cases	113
4.5	Extending the Semantic Broker.....	116
4.6	Summary	118
5.	Conclusions and Recommendations	119
5.1	Introduction	119
5.2	Summary of the Research	119
5.3	State of CERTCORT.....	121
5.4	Future Research Recommendations.....	122
5.4.1	Developing a Builder Agent.....	122
5.4.2	Extending the Signature Concept.....	123
5.4.3	Extending the Semantic Broker's Transformation Capabilities	123
5.5	Summary	123
Appendix A.	Selected Source Code	125
A.1.	Component Generation	125
A.2.	Relevant Component Retrieval.....	129
A.3.	Component Transformation.....	135
Appendix B.	Metadata.....	146
Bibliography	153
Vita	155

TABLE OF FIGURES

Figure 1:	CERTCORT Vision	3
Figure 2:	CERTCORT Agent Framework.....	4
Figure 3:	SUPPRESSOR Scenario Creation and Execution.	9
Figure 4:	SUPPRESSOR Player Structure.....	11
Figure 5:	Available System Types.....	12
Figure 6:	Input Parameters to Tactics	13
Figure 7:	MSFD Data Record Format	14
Figure 8:	SUPPRESSOR Player-Structure Object Representation	15
Figure 9:	SUPPRESSOR TDB File Format.....	16
Figure 10:	UML Class Diagram Derived from SUPPRESSOR Syntax	17
Figure 11:	Scenario as an Aggregation of Database File Classes.....	18
Figure 12:	MSFD UML Diagram.....	20
Figure 13:	MSFD Command Chain Representation.....	21
Figure 14:	Example SysTables System Table.....	22
Figure 15:	Example SysColumns System Table	22
Figure 16:	RMMS Architecture.....	28
Figure 17:	The acceptVisitor method	29
Figure 18:	Use of SAV to Analyze Object Tree	30
Figure 19:	JTree Terminology.....	31
Figure 20:	Scenario Component Representation	47
Figure 21:	Sample Scenario Component Representation	49
Figure 22:	SemanticGateway Component Generation.....	50
Figure 23:	Sample MetaSyntaxUnit Definitions	53
Figure 24:	Signature Analysis Approach	54
Figure 25:	Level of Abstraction Tradeoff	56
Figure 26:	Translation Categories of Data Items.....	58
Figure 27:	Application-Level View of Semantic Broker Architecture	60
Figure 28:	ScenarioRegistry Application Main Components and Data Sources	61
Figure 29:	Component Analysis	64
Figure 30:	SemanticGateway Application's Major Components	65
Figure 31:	SemanticGatewayAgent Conversation Process	67
Figure 32:	Component Transformation Classes and Data Source	70
Figure 33:	Transform Metadata File Format.....	71
Figure 34:	Semantic Broker Scenario Component Class Hierarchy	74
Figure 35:	Signature Analysis Classes.....	79
Figure 36:	Semantic Broker System-Level View	82
Figure 37:	ScenarioRegistry Application Class Diagram	83
Figure 38:	ScenarioRegistryAgent Class Hierarchy	85
Figure 39:	Scenario Registry Data Class Diagram.....	86
Figure 40:	ScenarioRegistryGUI Window.....	88
Figure 41:	SemanticGateway Application Class Diagram	90
Figure 42:	SemanticGateway Application	91
Figure 43:	SemanticGatewayAgent Class Diagram	93
Figure 44:	Source Registry Class Diagram	94
Figure 45:	Source Registry Graphical User Interface	96
Figure 46:	SignatureSelector Window.....	98
Figure 47:	SignatureSelector Popup Menu	99

Figure 48:	SemanticGateway Application with Signature Selected	100
Figure 49:	SemanticGateway Window After Relevant Component Retrieval Process	101
Figure 50:	SemanticGateway: Retrieve Component Details Menu	102
Figure 51:	SemanticGateway: Component Details Expanded	103
Figure 52:	Transformation Sub-System Class Diagram	105
Figure 53:	TransformEngine Class xFormModel and xFormComp Methods	106
Figure 54:	Component Transformation Process.....	108
Figure 55:	Transformation Menu of the SemanticGateway	109
Figure 56:	ComponentViewer Window.....	110
Figure 57:	Transformed Component with Untranslatable Components	111
Figure 58:	SemanticGateway – ScenarioRegistry Interaction	112
Figure 59:	Homogeneous Test Configuration.....	114
Figure 60:	Heterogeneous Test Configuration	115
Figure 61:	Research Impact on CERTCORT Agent Framework	120
Figure 62:	State of CERTCORT Functionality.....	121

ABSTRACT

The Department of Defense utilizes various simulation systems to model employment of forces and weapons systems in operational environments. The data files that model these environments and weapons systems are extremely large and complex, and require many person-hours to develop. Compounding the problem, these data files are distributed across multiple systems in a heterogeneous environment. Currently, there is no automated means of identifying and retrieving reusable portions of these files for reuse in a new scenario under development. This work develops a multi-agent system that catalogs the files, and provides the user with a means of identifying and retrieving reusable components. Additionally, since the format of the source files varies from simulator to simulator, a process for performing scenario component transformation is developed and implemented.

A SEMANTIC INTERFACE TO SCENARIO COMPONENT REUSE IN DOD SIMULATION SYSTEMS

1. INTRODUCTION

The Department of Defense (DOD) uses simulation systems to provide realistic, cost-effective training to enhance personnel readiness without needlessly jeopardizing their safety. Modern simulators generate complex training scenarios that would be cost prohibitive if conventional training techniques were employed. The DOD utilizes training systems which provide rehearsal of missions involving land, sea, air, and space based forces. The Air Force Research Laboratory Sensors Directorate, Electronic Warfare Simulation Branch (AFRL/SNZ) develops and maintains several such mission-level scenario systems. These systems include the Extended Air Defense Simulation (EADSIM), Suppressor Composite Mission Simulation System (SUPPRESSOR), Joint Interim Mission Model (JIMM), and the Simulated Warfare Environment Generator (SWEG). Previous research [Col99, Web99, Str99] focused on interoperability and reusability of model components. Captain Todd McDonald [McD00] researched the utilization of an extensible multi-agent framework to enhance the aforementioned functionality. This work employed agent technology to map SUPPRESSOR scenario files into syntactically correct object-oriented data structures. Captain McDonald's efforts provided an agent-based frame-work that provides a solid foundation for further exploration of the potential benefits of employing agent-based technology to provide

scenario component retrieval, transformation, and reuse. This present research investigates techniques for extracting suitable components from existing scenarios and presenting them to the user for reuse in a new scenario.

1.1 Definition of Terms

In order to avoid confusion and reduce ambiguity, a definition of key terms used in this research effort is provided. The key terms as used in the context of this document are as follows.

- **Model:** An object-oriented class hierarchy.
- **Scenario Component:** A system instance, e.g., an airplane, tank, building, etc.
- **Scenario:** A data repository that contains one instance of a simulation with specific player information.
- **Scenario Database:** A data repository that contains scenarios.

1.2 Problem Statement

The Air Force Research Laboratory, Sensors Directorate, Electronic Combat Branch (AFRL/SNZW) employs a set of databases to generate scenarios for its simulation systems. The Collaborative Engineering Real Time Database Correlation Tool (CERTCORT) is under development by AFRL to provide an interoperability bridge between the various simulation systems. Figure 1 depicts the CERTCORT concept.

The ultimate goal of CERTCORT is the reusability of an existing scenario, or portion thereof, in the creation of a scenario for a different type of simulator. AFRL is interested in developing a more efficient means of scenario generation. For example, consider the situation where an analyst is building a new JIMM scenario. There is an existing SUPPRESSOR scenario that meets most or all of the analyst's requirements except, of course, its format is not JIMM. The CERTCORT system will ultimately be capable of

assisting the analyst in identification of the existing SUPPRESSOR scenario and transformation of the scenario to JIMM format.

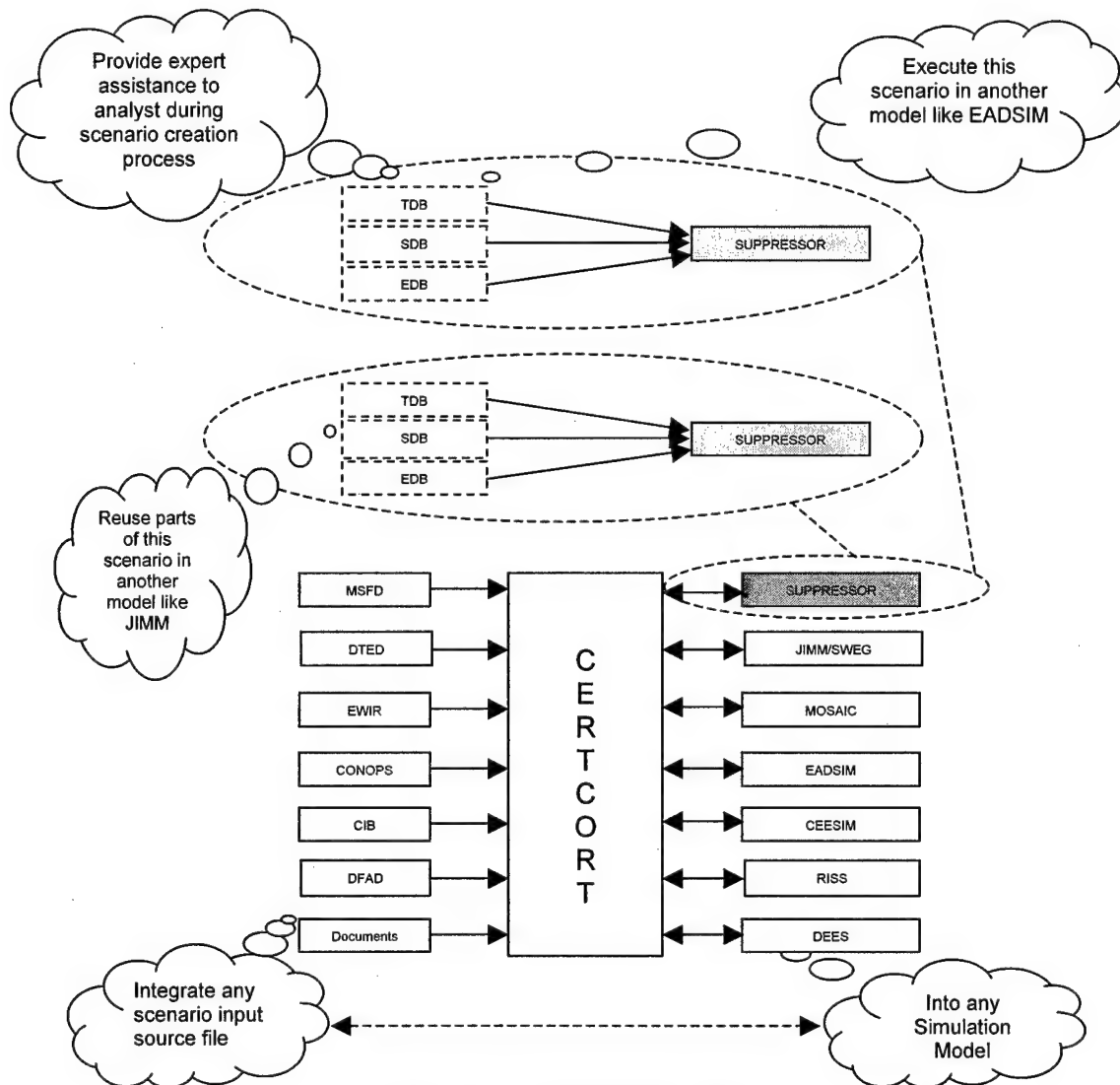


Figure 1: CERTCORT Vision

Since the current system does not provide the facilities necessary to identify existing, relevant scenarios, users require extensive knowledge of the underlying scenario database. These expert users must manually search the volumes of text-based scenario source files, and utilize their extensive knowledge to identify certain characteristics that determine a scenario's composition. Utilizing various identifying

characteristics to determine the content of a given scenario component is known as semantic interpretation. Semantic interpretation involves asking questions about a given scenario component, such as:

- What is the type of the scenario player?
- What type of systems, capabilities, and susceptibilities are attached to the scenario player?
- Is the subject scenario component similar to an existing, known component?

There is currently a prototype multi-layered agent-based system in place, McDonald [McD00], that retrieves data from SUPPRESSOR text files and instantiates an object model. The system uses software agents, acting as information requestors, brokers, and providers to supply the user with available scenarios in their entirety. Figure 2 [McD00, 209] provides a graphical representation of the CERTCORT agent layers.

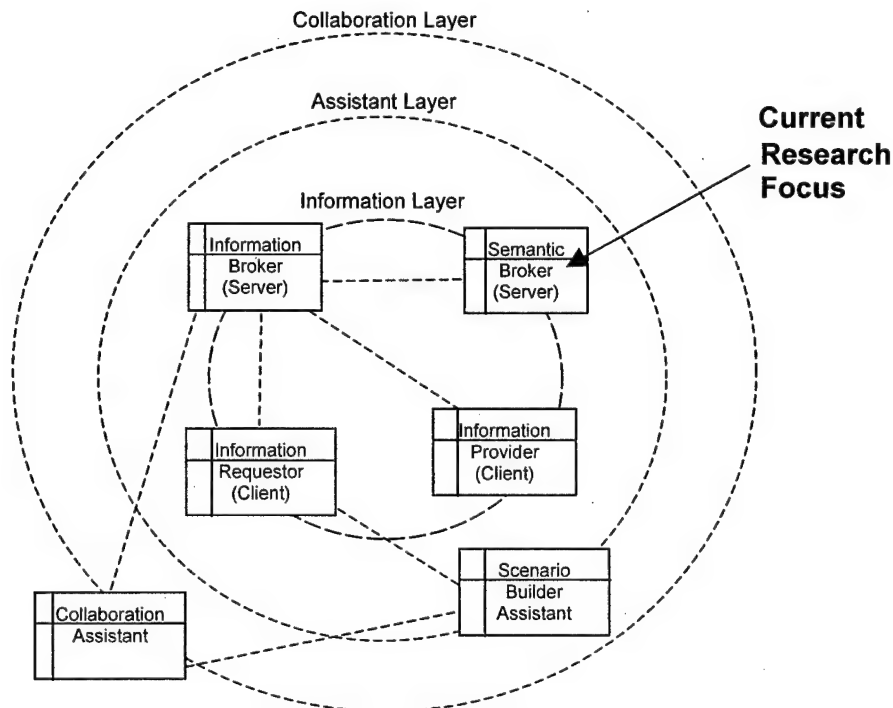


Figure 2: CERTCORT Agent Framework

However, the object model developed by McDonald is very specific to the SUPPRESSOR scenario type and does not facilitate the representation of other scenario formats (e.g., SWEG, JIMM, etc.). This aspect of the current system reduces scenario piecewise reusability. The current system focuses almost entirely on the syntactical aspects of the scenario files—how the files are structured, but not what they represent (i.e., the semantics). Semantic information conveys what a scenario component represents and can only be determined by evaluating the entire scenario component. Semantic information, what a component is and does, can be ascertained by analyzing the component and noting what sub-components it contains and the characteristics of those sub-components. Captain McDonald's research effort excluded the semantic broker agent. AFRL desires that the system be capable of presenting multiple levels of detail about a scenario's components as well as providing a more coherent and reusable representation of those details. For example, if the analyst requests an F-16 as part of building a simulation, the semantic broker should know which scenario types contain F-16 aircraft, the original source of the scenario components, and their composition.

1.3 Research Focus

The design, development, evaluation, and insertion of a semantic broker in the information layer of the CERTCORT multi-agent architecture will be the primary focus of this research effort. As stated previously, much work has been accomplished toward developing a common object model for the various simulation types; however, research on the semantic agent aspect is only now coming into focus. For the purposes of this research, the semantic agent's responsibilities include:

- 1) Presenting the user with an interface suitable for describing the user's data requirements.
- 2) Maintaining and querying the appropriate data structures.

- 3) Presenting the user with the existing scenario components that match the requirements.
- 4) Providing facilities to transform a selected existing scenario component to the desired scenario format.

In order to successfully retrieve scenario components for a user, the system must determine as precisely as possible what data the user wants. The most obvious manner of specifying data requirements to the system is through a semantically enriched user interface. The current interface does not allow the user to present requirements that are sufficiently detailed to permit the semantic agent to extract the required data. The semantic agent must provide a means, most likely through a user interface, of determining user information requirements.

After the user's information requirements have been solidified, the semantic agent must query existing scenarios to determine if the requested data is present in the currently available data sources. These sources may include scenario source files belonging to any of the simulators in the CERTCORT system, as well as the native input sources to developing these simulator scenarios such as the Multi Spectral Force Deployment database.

Finally, after the data has been gathered, it must be presented to the user in a format that facilitates comprehension, traceability to source, and ultimately, reusability. In the previous example of the request for an F-16 scenario component, the returned scenario components can be displayed, for example, in a Java-based tree structure.

1.4 Summary

The Air Force Research Laboratory, Sensors Directorate, Electronic Combat Branch (AFRL/SNZW) employs a set of databases to generate scenarios for its simulation systems. AFRL is developing the CERTCORT system to integrate the various

databases and facilitate the migration of scenarios from one simulator format to another. However, while the current system is capable of locating existing scenario components, it does not present sufficient semantic content to permit the user to determine whether it is appropriate for inclusion in a new scenario. This new research investigates methods for finding and retrieving existing scenario components, presenting them to the user, and preparing selected components for inclusion in a new scenario. Specifically, the primary focus of this research is the responsibilities of the semantic broker in the CERTCORT Agent Framework depicted in Figure 2.

2. LITERATURE REVIEW

2.1 Introduction

There are several technologies central to understanding the magnitude of the problems involved in providing enhanced component semantics to simulation scenario creators. Since the primary focus of this research effort involves the SUPPRESSOR simulator, the structure of the SUPPRESSOR scenario files and the various input sources to the creation of these files are analyzed. One specific input to scenario creation, the Multi-Spectral Force Deployment (MSFD) database, is explored here to examine its structure. The CERTCORT class hierarchy is also reviewed to determine appropriate extensions to enhance semantics of the model. The role of metadata, what it is and how it can be exploited to extract meaning from a scenario component or group of components is also examined. Additionally, the *visitor* design pattern is explored to uncover its capabilities and potential use in development of the semantic broker's analysis engine. Next, the Java Foundation Class component JTree is scrutinized to discover its capabilities and nuances. Since this work involves extending the CERTCORT agent framework developed by McDonald [McD00], the peculiarities of layered, multi-agent systems are also extremely relevant to this research. Therefore, the applicable areas of agent technology are covered next. The CERTCORT system utilizes the *ObjectStore* database to make its objects persistent, so a review of object-oriented databases is in order. Since the source files used to generate a specific scenario are diverse in both format and content, the constraints and issues specific to heterogeneous Multi-Database Systems (MDS) factor into this effort and are the final topic of this literature review.

2.2 Suppressor Overview

Suppressor is a digital computer model, general-purpose simulation of a possibly multi-sided conflict involving some combination of air, ground, naval, and space-based forces [SAIC97]. To understand the overall functionality of the Suppressor Composite Mission Simulation (SUPPRESSOR), a review of the SUPPRESSOR scenario creation

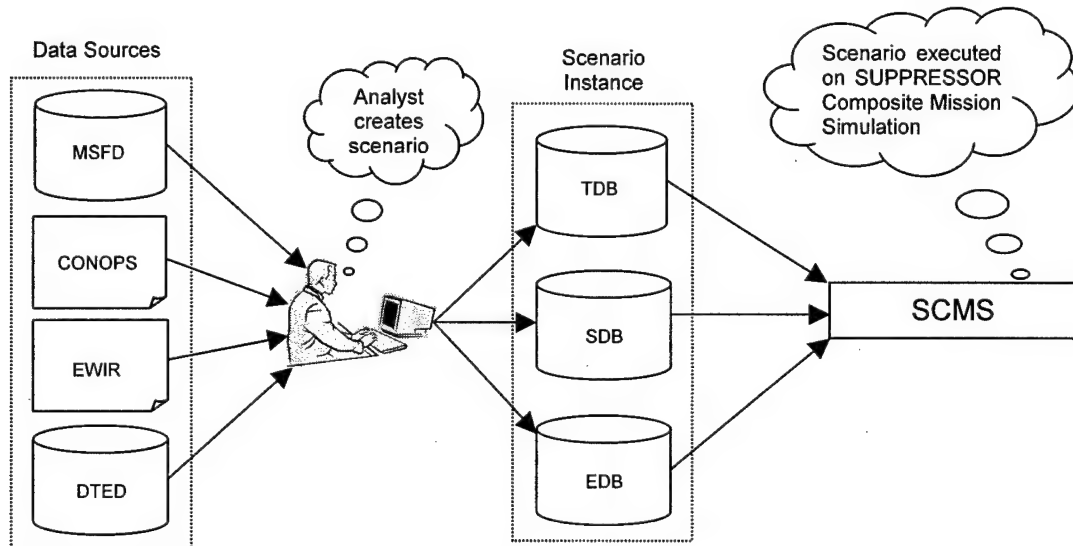


Figure 3: SUPPRESSOR Scenario Creation and Execution.

process, the key data sources, and the resulting scenario files is essential. The SUPPRESSOR scenario generation process is depicted in Figure 3. There are several data source input files used to develop a SUPPRESSOR scenario. These source files may include all of the following:

- **Multi-Spectral Force Deployment (MSFD):** A file containing player and system locations and the command hierarchy.
- **Digital Terrain Elevation Data (DTED):** A flat file containing a digitized representation of the geographical area in which the simulation will occur.
- **Electronic Warfare Integrated Reprogramming (EWIR) Database:** A document containing the electronic signature parameters and performance capability of weapon systems.

- **Concept of Operations (CONOPS):** Any document that contains tactics and doctrine for any given force deployment to ensure accurate representation of players and systems in the scenario.

As shown in Figure 3, the analyst uses these data sources to create players and systems, detail geographic attributes, define threat parameters and weapons capabilities, and ensure current doctrine and tactics are being followed. The work of the analyst results in the following files [SAIC97, 1-6], which comprise a suppressor scenario:

- **Type Database (TDB):** Provides a mechanism for the user to describe data that is shared by types of players, elements, and systems; or shared across the board by more than one type of player, element, or system.
- **Scenario Database (SDB):** Contains information specific to each player, such as its location, movement path, engagement zones, communication nets, etc.
- **Elevation Database (EDB):** Provides the ability to access Defense Mapping Agency (DMA) terrain data, transform it, and use it during model execution for line of sight checks for sensors, communications, and jammers.

The TDB contains most of the user's data, and can be used for more than one SDB. In fact, usually there will be one TDB per study, with several SDB's for each important variation within the study [SAIC96, 3-2]. Additionally, a User Application Names (UAN) file contains the study-specific collection of names that allow the user to converse with the model [SAIC97,1-6].

An interesting feature of SUPPRESSOR is the fact that it requires user provided instructions that tell it how to interpret the user data. As stated in Volume I of the Suppressor Release 5.4 User's Guide, "Suppressor requires the user to study the problem and prepare a set of input instructions to represent the problem or situation." The user provides data item definitions that adhere to the syntax of the SUPPRESSOR language. By applying the parsing rules of this very specialized grammar, SUPPRESSOR is able to run and execute the simulation.

The TDB and SDB files define instructions and data values for player structures. A player can be represented at any level of detail, and consists of the following organizational components as shown in their hierarchical sequence in Figure 4 [SAIC97, 2:52].

- **Locations:** A player can have one or more locations, and a location can be moving or stationary.
- **Elements:** An element is a susceptible component of a player. An element belongs to a location and a location has one or more elements. Each element has signatures that reflect how it will be perceived by sensors. Elements have *susceptibilities*.
- **Systems:** An element may have one or more systems. These systems give the player the ability to perform specific functions. There are eight systems that elements may have as outlined in Figure 5 [SAIC97, 2:53].
- **Resources:** Some systems have explicitly modeled resources such as bombs, missiles, etc.
- **Tactics:** Define how a player will use the systems to act on its own and interact with other players. Players with no systems have no tactics. There are five categories of tactics input parameters as listed in Figure 6 [SAIC97, 2:54].

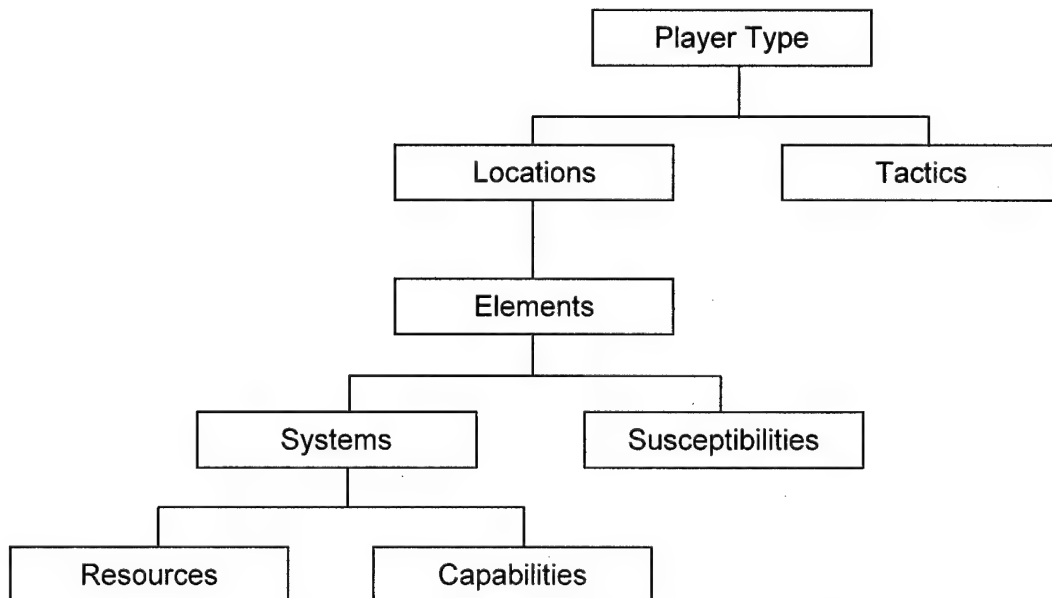


Figure 4: SUPPRESSOR Player Structure

System	Usage	Generic Function
Sensor Receiver	Allows players to noncooperatively gain information on other players; can be nonlethally engaged by a disruptor system; covers four types of sensors: radar, radar warning, infrared, and optics receivers.	SENSE
Sensor Transmitter	Used in conjunction with radar sensor receivers, allowing players to noncooperatively gain information about other players.	SENSE
Communication Receiver	Allows players to receive messages from other players; when linked with a communication transmitter, there can be two-way dialogue; can be nonlethally engaged by disruptor systems.	TALK
Communication Transmitter	Allows players to send messages to other players.	TALK
Weapon	Allows players to lethally engage other players.	SHOOT
Disruptor	Allows players to nonlethally engage other players and affect their ability to gain information.	DISRUPT
Thinker	Processes information based on input tactics and intelligence data received and simulates the processing time required to make these decisions.	NOTICE DIGEST REACT
Mover	Provides capability for a location to change its position over time; movement can be preplanned, and in conjunction with a thinking system, reactively modified.	MOVE

Figure 5: Available System Types

A sensor receiver and, if applicable, sensor transmitter system can simulate one of four types of sensors [SAIC97, 2:53-54]:

- Radar: active sensors which give off energy and reflect that energy back; requires linkage of sensor receiver system and sensor transmitter system; detects elements.
- Optics: passive sensors that work off energy emitted by the target from something shining on it; requires only a sensor receiver system; detects elements.
- Infrared: passive sensors that work off energy emitted by the target; requires only a sensor receiver system; detects elements.

- **Warning Receiver:** picks up emissions from communication transmitter and sensor receiver systems; requires only a sensor receiver; detects systems, not elements.

Lethal Assignment	characterizes the interrelationships between commanders and subordinates in making and receiving assignments
Lethal Engagement	defines the guidelines by which players with weapons will engage threats
Non-Lethal Engagement	defines the guidelines by which players with disruptors can engage threats
Coordination	a multifunctional area encompassing tactics associated with intel reporting, engagement requirements and zone permissions
Movement	defines the guidelines for maneuvering vehicles including those related to launch and terrain following, terrain avoidance, threat avoidance, and contingency planning

Figure 6: Input Parameters to Tactics

Players can be defined at varying levels of detail, each with different data requirements. The level of aggregation and detail used to describe a player will influence the amount of data required to represent the player types, the way command, control, and communications between players in the scenario is defined, and the interrelationships between players [SAIC97, 2:54].

Automating the scenario creation process is a goal central to the CERTCORT vision. Some work has been done on a translation program that automatically generates a partial SUPPRESSOR SDB from a multi-spectral force deployment (MSFD) file. MSFD is investigated in the next section.

2.3 Multi-Spectral Force Deployment (MSFD)

The MSFD contains information detailing the subordination relationships of units from the national command level down to the company and battery levels. MSFD

provides the necessary data for command-level force structure analysis. The format of MSFD files is one 14-element record per line. These 14 elements are enumerated in Figure 7.

DATA ELEMENT	COLUMN(S)
Source Identifier	1
Sequence Number	2-8
Record Type	9
Blank	10
Site/Equipment Name	11-21
Location Lat/Long	22-38
Site Function	39-40
Location UTM	41-53
Unit Subordination Code	54-64
Map Scale	65
Spheroid	66
Site Status	67
Time Frame	68-72
Comments	73-80

Figure 7: MSFD Data Record Format

Given this format, MSFD is considered a scenario data input to SUPPRESSOR rather than a scenario database file. All SUPPRESSOR scenario database files conform to the SUPPRESSOR grammar rules. The MSFD file's records correspond and can be mapped to equivalent structures within SUPPRESSOR's SDB file with minimal transformation.

Previous research ([Col99], [Web99], [McD00]) has focused on developing a common object model for the Collaborative Engineering Real Time Database Correlation Tool (CERTCORT). In all of the above mentioned efforts, developing object models for SUPPRESSOR and MSFD and integrating them with other CERTCORT data models was a main focus. The most current research in this area was performed by McDonald ([McD00]), and his object models for SUPPRESSOR and the MSFD database will be investigated next.

2.4 CERTCORT Common Object Model

In developing an object model for SUPPRESSOR, McDonald built on previous work by Weber [Web99]. McDonald extended Weber's object model and developed a parser that maps textual player-structure definitions to player object models. This model is shown in Figure 8 [McD00, 117].

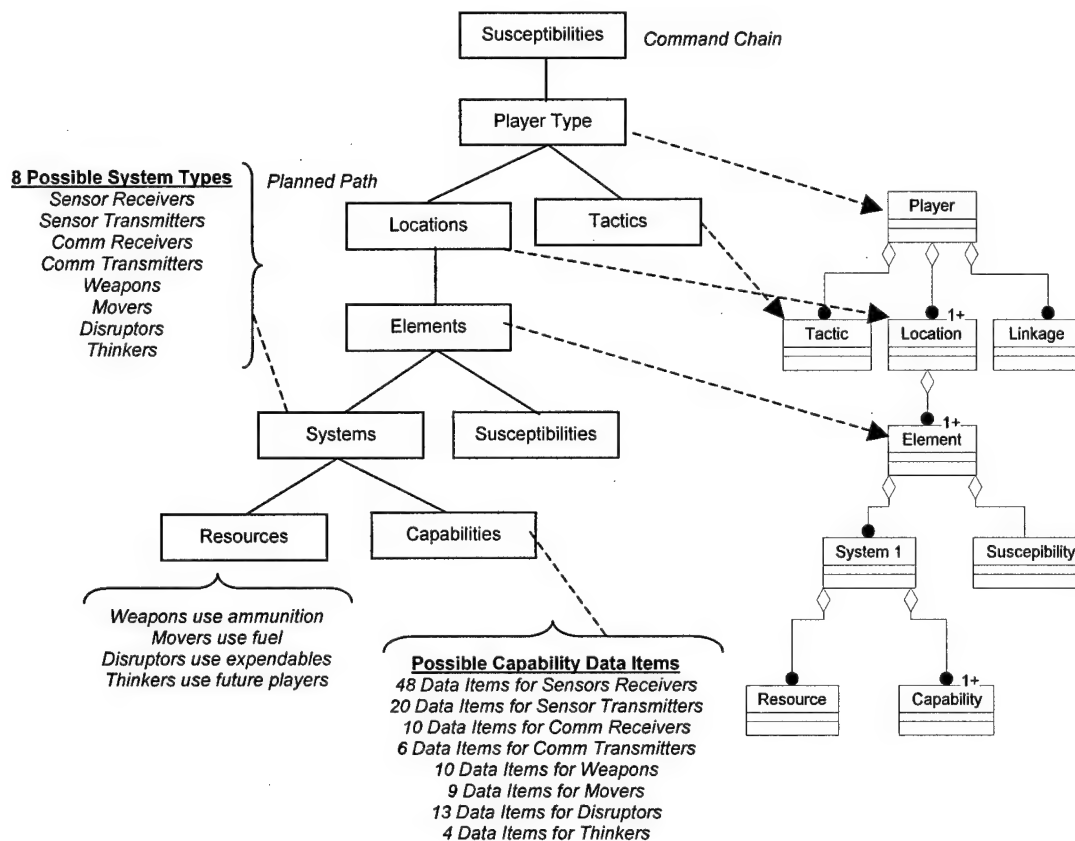


Figure 8: SUPPRESSOR Player-Structure Object Representation

2.4.1 SUPPRESSOR Grammar Parser

McDonald also developed an object model for the SUPPRESSOR grammar. A grammar is made up of keywords, tokens, and production rules. A production rule defines a precise, allowable order for a sequence of keywords and tokens. Each rule,

with the exception of the top-level rule, is an expansion of a previous rule in the previous level. Since there are a finite number of possible expansions from one level to the next, a grammar parser is able to read a file and determine if the text conforms to its grammar rules. These grammar rules are also known as the language's syntax. The top-level production rule in SUPPRESSOR consists solely of the "EXECUTE" keyword. Each of SUPPRESSOR's input files (TDB, SDB, and EDB) conforms to the SUPPRESSOR syntax. The structure of the SUPPRESSOR TDB input file is given in Figure 9 [SAIC97].

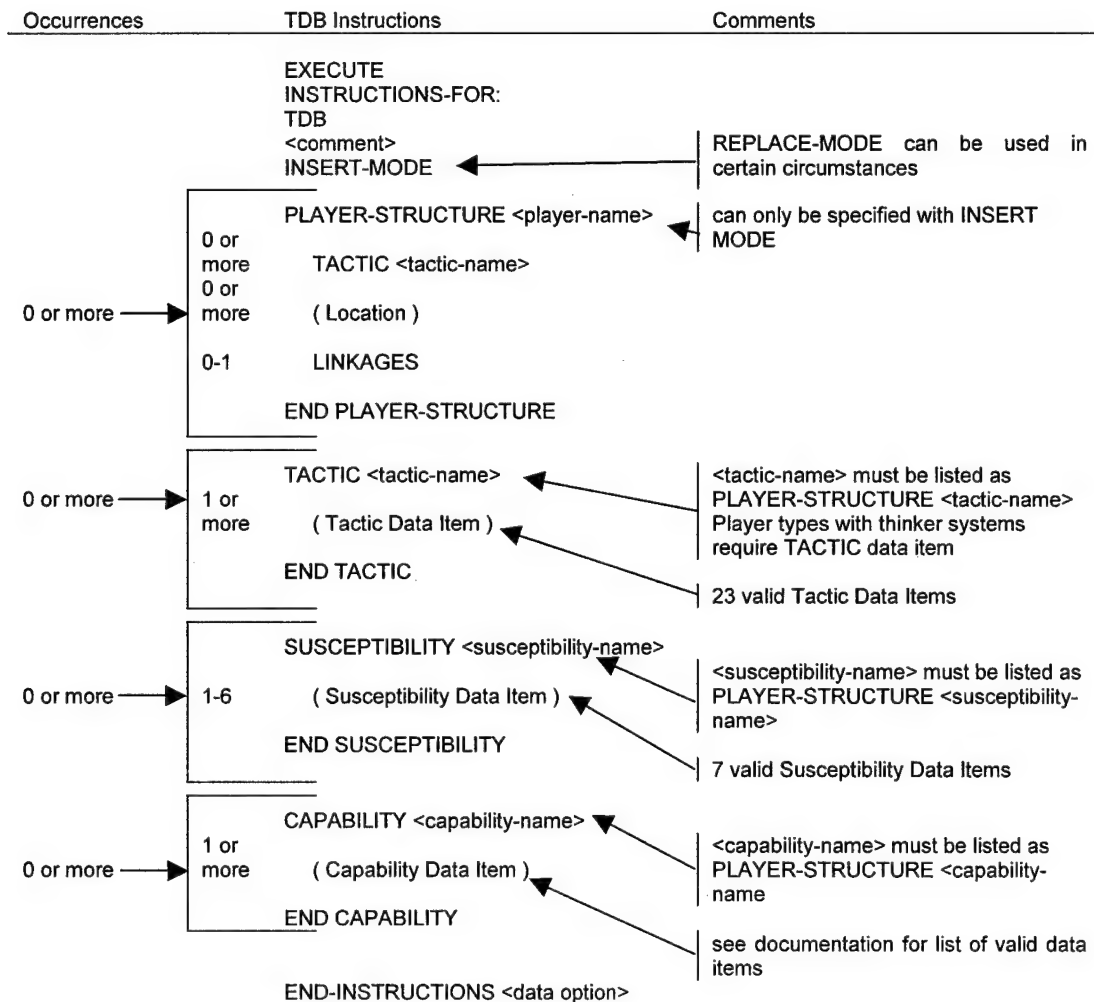


Figure 9: SUPPRESSOR TDB File Format

The format of the SDB instructions, although not identical, follows a nested pattern similar to the TDB file format. Based on the SUPPRESSOR language syntax, McDonald

derived the UML class diagram shown in Figure 10. In the figure, the Boolean attribute *saveData* of the *ExecuteCompositeltem* class relates directly to the *<data-option>* in the grammar syntax, and its value determines whether scenario execution results are saved or discarded. The Boolean attribute *hasUI* provides the capability to use the class independent of, or in unison with, a graphical user interface (GUI). At implementation, a pointer to a GUI can be found in the *userInterface* attribute of *ExecuteCompositeltem* if the instance has been created by a user interface [McD00, 123].

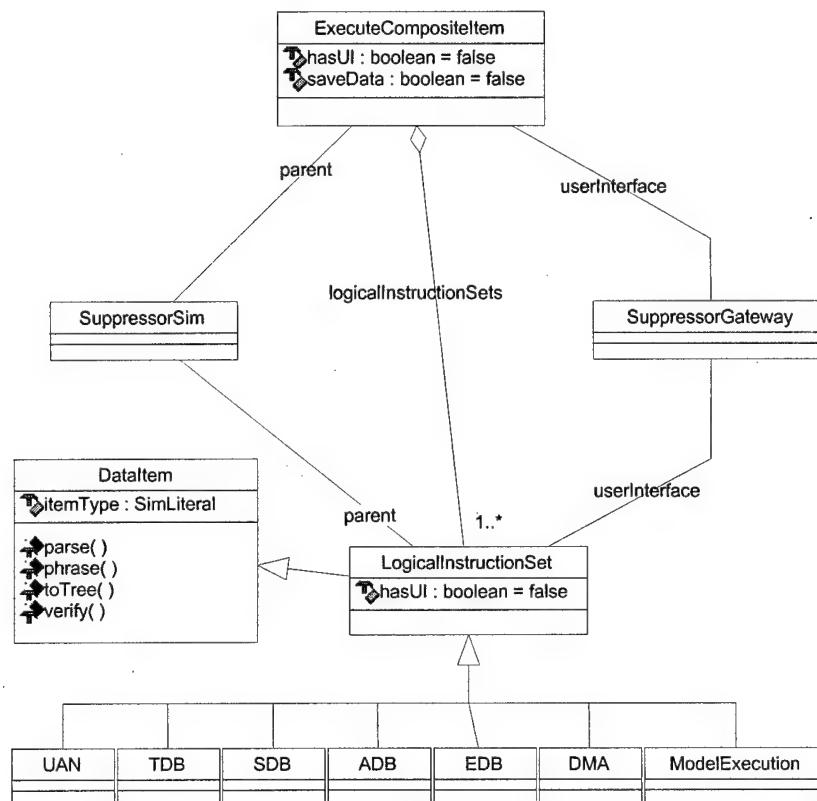


Figure 10: UML Class Diagram Derived from SUPPRESSOR Syntax

2.4.2 SUPPRESSOR Object Model

A SUPPRESSOR scenario is composed of various files that define the players, tactics, environment, etc. for a simulation. These files are aggregations of definitions of the components and subcomponents that detail the scenario's characteristics. Figure 11

[McD00, 149] depicts the CERTCORT UML class hierarchy that represents these files and their components of a SUPPRESSOR scenario.

In Figure 11, the UAN class represents the user application names set of instructions used in the SUPPRESSOR model. The UAN class is composed of a collection of definitions relevant to the scenario(s) being studied. These definitions each identify a specific group of entities according to their usage in the TDB and SDB [SAIC, Vol II: 3-2].

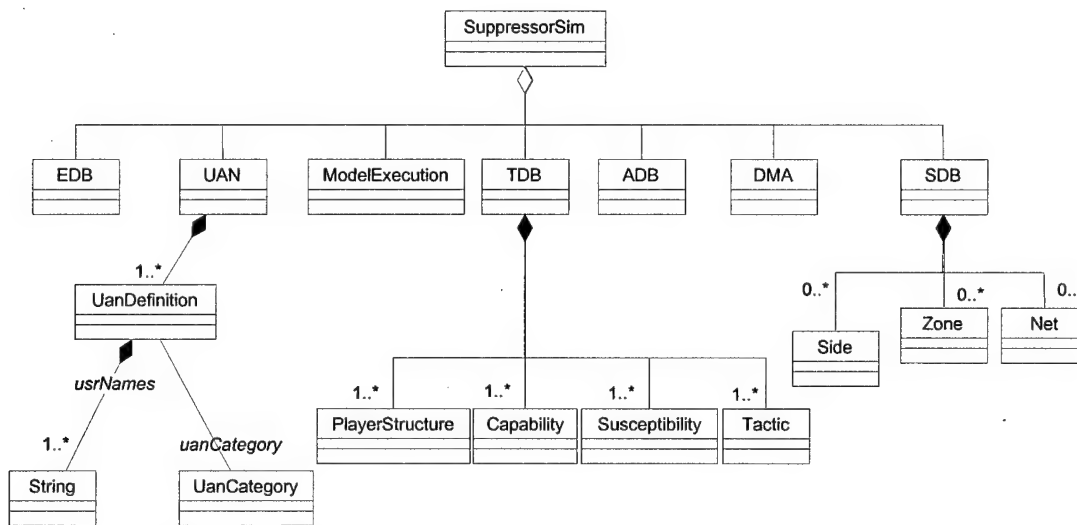


Figure 11: Scenario as an Aggregation of Database File Classes

Based on Figures 9 and 11, one can discern the data elements that make up a PLAYER-STRUCTURE in a TDB file. A TDB file consists of zero or more definitions of the following types: PLAYER-STRUCTURE, TACTIC, SUSCEPTIBILITY, and CAPABILITY. Strictly speaking, the definition types TACTIC, SUSCEPTIBILITY, and CAPABILITY cannot exist without at least one PLAYER-STRUCTURE definition. The PLAYER-STRUCTURE is an aggregation of TACTIC, LOCATION, and LINKAGES data types. The LOCATION data type is an aggregation of ELEMENT types. The ELEMENT type consists of SUSCEPTIBILITY and SYSTEM types, and the SYSTEM type is composed of CAPABILITY and RESOURCE types.

The scenario database (SDB) contains data that relates all the player descriptions found in the TDB to command chain structures, zones, and communication networks as well as other inter-player relationships [McD00, 160]. In addition to information detailing various attributes of the scenario, the SDB is composed of zero or more NET and ZONE components as well as one or more SIDE components. These components relate to communication networks, shared zones, and command structures respectively.

2.4.3 Multi-Spectral Force Deployment (MSFD) Object Model

In analyzing the MSFD file structure, McDonald [McD00, 173-174] notes that of the fourteen fields present in an MSFD record, there are six of primary significance to analyzing an MSFD record for use as an input source for a CERTCORT-based scenario:

- **Sequence Group:** A five digit number that begins with "00001" for the first unit in a sequence. The *sequence group* for a unit that is designated as any type of headquarters node or controlling authority will always be "00001" and those units authority will have the same *sequence group*, but a distinct and unique *sequence code*.
- **Sequence Code:** A two digit number that is assigned at "01" for all subordinate units that report to a given headquarters or controlling authority unit.
- **Record Type:** A single character field set to either "A" or "B." All headquarters and controlling authority units will be designated by an "A" and each subsequent record with a "B" entry indicates that it is subordinate to the closest preceding "A" record unit.
- **Site Equipment Name:** An eleven-character field that contains the site/unit/equipment name.
- **Geographic Location:** The latitude and longitude of the unit.

- **Unit Subordination Code (USC):** An eleven-character alphanumeric code that uniquely identifies and subordinates each site/unit in the data file. The USC reflects the command level structure from the highest national level to the lowest company level.
- **Time Frame:** A three-digit field that determines which year a unit is forecast at a given location. Scenarios typically use only units listed in the MSFD that have a common time frame. There are three possible positions for the time frame, indicated by the presence of an "X," which can map to three distinct years (i.e. 1995, 2000, 2010).

The UML diagrams in Figures 12 and 13 show the class hierarchies derived from the MSFD file record structure to represent the headquarters-subordinate relationships

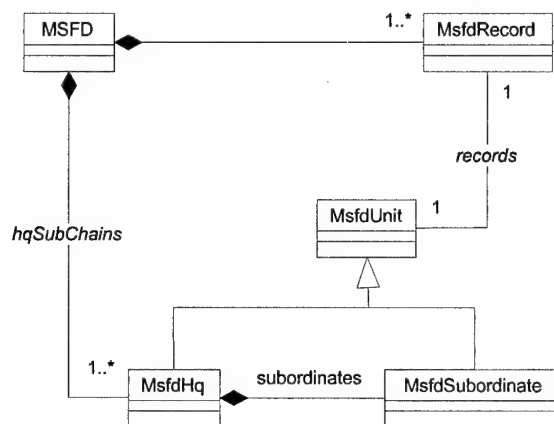


Figure 12: MSFD UML Diagram

and command chains respectively. Figure 12 defines two types of units for MSFD: 1) Headquarters units (*MsfdHq*), and 2) Subordinate units (*MsfdSubordinate*). This derivation of the MSFD format also results in a class named *MsfdRecord*, which is the entire collection of MSFD records from the MSFD input file. Therefore, an *MSFD* object is an aggregate of *MsfdRecord* objects, each of which has a one-to-one correlation to an

MsfUnit object. The class hierarchy in Figure 13 [Mcd00, 178] represents the seven command chain levels of the MSFD. The *MsfCommandChain* class provides a

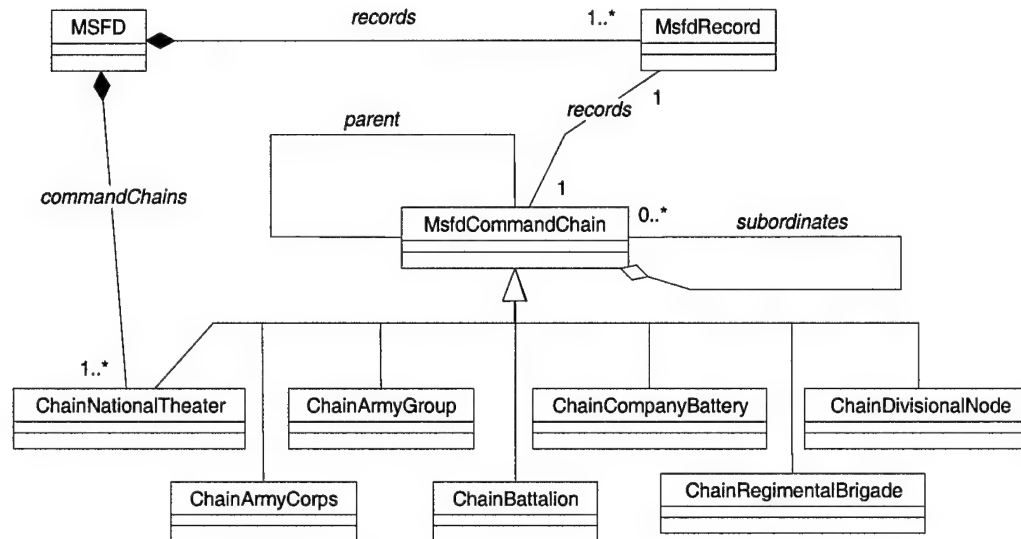


Figure 13: MSFD Command Chain Representation

generalized form of command authority representation, and adds the capability to store both parent and subordinate unit relationships [McD00, 178]. The seven subclasses of the *MsfCommandChain* class correspond directly to the six command authority levels of MSFD. The seventh subclass *ChainDivisionalNode* corresponds to units designated as division-level.

2.5 Metadata

A database is a self-describing collection of integrated records [Kro99, 14]. The portion of a database that contains this self-describing information is the data dictionary or metadata. Essentially, metadata is data about data. For example, if a particular column in a relational DBMS contains a numeric value representing an employee's age, the metadata concerning this column might be that it must be a positive integer. In a relational DBMS, this metadata is stored in special tables called system tables. Figures 14 and 15 provide examples of two types of system tables containing metadata.

Table Name	Number of Columns	Primary Key
Student	4	StudentNumber
Adviser	3	AdviserName
Course	3	ReferenceNumber
Enrollment	3	{StudentNumber, ReferenceNumber}

Figure 14: Example SysTables System Table

The table in Figure 14 contains a record for each table present in the database. These records store the number of columns of each table, and each table's primary key.

The table in Figure 15 contains the columns of every table in the database. This table specifies the table to which the column belongs, the column's data type, and its length. Depending on the application, it may prove useful to store additional information

Column Name	Table Name	Data Type	Length
StudentNumber	Student	Integer	4
FirstName	Student	Text	20
LastName	Student	Text	30
Major	Student	Text	10
AdviserName	Adviser	Text	25
Phone	Advisor	Text	12
Department	Advisor	Text	15
ReferenceNumber	Course	Integer	4
Title	Course	Text	10
NumberHours	Course	Decimal	4
StudentNumber	Enrollment	Integer	4
ReferenceNumber	Enrollment	Integer	4
Grade	Enrollment	Text	2

Figure 15: Example SysColumns System Table

in this table. For example, if there were a column that contained each student's weight, it would be useful to know whether the unit of measurement was pounds or kilograms. In addition to these two tables, there are system tables for indexes, keys, and other facets of the database structure.

Application metadata is another variant used to store the structure and format of user forms, reports, queries, and other application components [Kro99]. This metadata is created and updated by the DBMS's design tools subsystem when forms, reports, etc.

are created and modified; and utilized by its run-time subsystem when generating these components and linking them to data elements in the tables.

Traditionally, Database Management Systems (DBMSs) have not maintained other, more semantic data. For example, some data in the database may be derived from processing several databases. To build on the previous example of the column containing each student's weight, consider the following scenario. There are two separate and heterogeneous databases that must be processed and compiled into a third composite database containing all students. The first database contains U.S. students and maintains their weight attribute measure in pounds. The other contains European students and maintains the weight field measure in kilograms. In order to consolidate the two databases, a standard unit of measure must be adopted and a conversion process documented. This conversion process should be documented in the metadata of the derived database. However, the current generation of DBMS metadata facilities are not equipped to represent this conversion process. It was this lack of capability that led the Secretary of Defense to sponsor RAND research that led to development of the RAND Metadata Management System [Cam95].

2.5.1 RAND Metadata Management System

The RAND Metadata Management System (RMMS) is a system that manages metadata associated with the relational database management system operated by the Military Operations Simulation Facility (MOSF).

As motivation for developing RMMS, the RAND researchers enumerate several examples that exemplify the need for, and use of, metadata. A few of these are listed below [Cam95, 3].

- Social science studies integrating census data over many decades must be able to compare the schemas of different versions of data because through the years different data fields have been recorded. For example, census surveys early in the century asked if households had a flush toilet.
- Different environmental waste databases maintain contaminant levels differently, for instance, as concentration percentages or as a pair of weights representing solid waste and total waste. To compare contaminants across two such databases requires knowledge of the representations and conversion procedures each uses.
- Metadata is an ideal resource for browsing, making it possible to identify, for example, databases that contain information on military airfield and runway assets. Metadata serves to link references from standard data elements such as "airfield" and "runway" to the databases that contain data for these data elements.

As these examples illustrate, metadata is essential to heterogeneous database interoperability. The goal of a metadata management system is to centralize and standardize metadata information and associated procedures [Cam95, 4].

The RAND researchers addressed five major issues during development of RMMS. The first issue involved the need for complete, thorough, and standard data documentation. System manuals, when provided, are useful for the system administrator, but are of little use to users who want to know the content of the database. Often there is no formal documentation regarding the organization and semantics of the database.

The second issue raised was the need to record and manage information about different versions of databases. In organizations that use databases generated from

outside sources, the issue of what to do with the old version when new ones arrive must be resolved. If several versions of the database are required to be available, a means must be implemented to store these old versions and procedures must be adopted to ensure their compatibility with any future update to the associated DBMS.

The need to maintain a history of the changes made to database tables, schema, and data values was also addressed in the RAND effort. The metadata management system should be capable of maintaining a history of data values and schema changes. This would permit the re-creation of previous versions of the database.

The fourth issue addressed by RAND was the need to facilitate derived databases for input to simulation models and for sharing among models. A metadata management system should facilitate generation and storage of metadata for a derived database. This generation and storage should be performed automatically when a derived database is created, and should include the sources and process used in the derivation.

The last issue raised by the RAND researchers was the need to standardize the names of data elements that are (1) conceptually the same but named differently or (2) named the same but conceptually different. This issue is directly related to interoperability conflicts among heterogeneous data sources. The metadata management system must be able to resolve the naming issues and perform conversions on any data fields that, for example, have the same meaning, but are represented in different units of measure.

Most commercial DBMSs include a repository called a data dictionary. The data stored in this repository is generated during schema creation and is generally limited to those characteristics specified in the data definition language [Cam95, 9]. These characteristics normally include items such as table name, number of columns, primary key, and owner name. A column table may contain data on the column name,

associated table, data type, length, and whether or not nulls are permitted. Other information is also stored in the data dictionary, but most of this information is only relevant for the operation and optimization of the DBMS [Cam95, 9]. This repository cannot be used to store semantic information such as units of measure, procedures for conversion, or in the case of a derived table, the sources and derivation process.

RMMS uses conventional relation DBMS techniques to deal with the various issues previously discussed. The RMMS approaches the documentation problem by providing the capability to store information about databases, tables, and column entities. RMMS augments the DBMS provided system tables with inter- and intra-table relationships, attribute domain information, and aliases.

RMMS addresses the history issue by recording all changes made to a database. When modifications are made the pre-change value is recorded in a metadata table. There are three "history" metadata tables as defined below [Cam95, 12].

- Value History Table: store old values from data tables that have been subsequently updated
- Table History Table: store changes to a table as a whole (i.e. a name change)
- Column History Table: store changes to the schema

To facilitate database derivation, RMMS maintains a specialized metadata relevant to derived databases, procedures to automate the process of deriving metadata from external database metadata, and a trigger mechanism for automatically updating derived databases when an external source changes. To achieve these functions, RMMS provides a registry of external data sources, which is linked to metadata on those sources.

Data element standardization was achieved through the use of aliases and conversion procedures. RMMS defines a standard name for each data element, then

links these to the various aliases and necessary conversion procedures. There are two benefits to this approach: 1) When creating derived databases, the new columns can be named using the standard element names, which reduces ambiguity; and 2) When performing queries, users can refer to the standard element name and obtain references to the various aliases. This approach maintains the independent structure of the heterogeneous data sources.

The RMMS architecture, depicted in Figure 16 [Cam95, 16], has two major components. The first is called the "Data Encyclopedia." The data encyclopedia is a database of metadata concerning all application databases. There is only one data encyclopedia for all the application databases. The other component is the "Data Dictionary," which is a set of metadata tables that augments the system tables for each application database. There is one data dictionary for each application database. Figure 16 shows these two components and their relationship to the DBMS and application databases. In the figure, "sde" is an acronym for "*standard data element*," and "md" is an abbreviation for "*metadata*."

Although the RMMS system was developed for use with a relational DBMS, the basic principal of using metadata to provide interoperability among heterogeneous data sources and enhance the semantic content of the overall system can be applied to object-oriented DBMSs and, to a lesser extent, flat file systems as well. This research will further investigate the possible use of these techniques to enhance the representation of scenarios and their individual components.

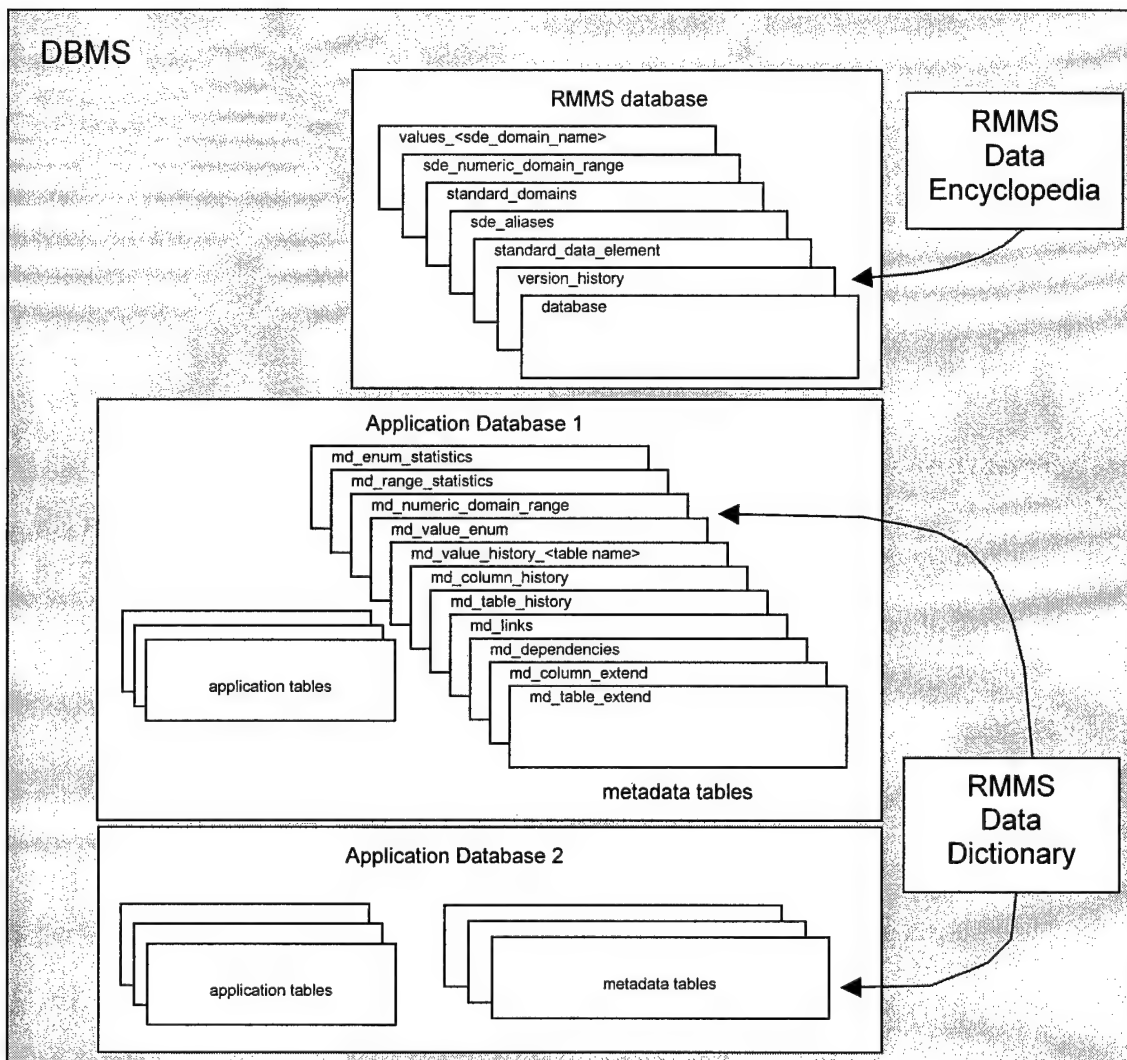


Figure 16: RMMS Architecture

2.6 The Visitor Pattern

The *visitor* design pattern has been described in Design Patterns: Elements of Reusable Object-Oriented Software [Gam95]. Visitor allows the addition of new operations without modification of the class of elements on which it performs. The visitor pattern provides a framework for packaging related operations into an object separate from their classes--a visitor object. Each class then implements the *acceptVisitor* method shown in Figure 17. When an object accepts a visitor, it calls the *visit* method of

the visitor object and passes itself as a parameter in the method call. The visitor class defines a method for each class of objects it knows how to analyze.

```
public Object acceptVisitor (Visitor visitor, Object data)
{
    return visitor.visit(this, data);
}
```

Figure 17: The acceptVisitor method

By employing visitor, all methods needed to perform semantic analysis on the over one hundred CERTCORT object classes could be located in one file—controlled by the semantic broker. This is accomplished by creating, through either inheritance or an interface, a visitor class. This class contains a semantic analysis method, a.k.a. a visitor, for each target class to be analyzed. All the code required to analyze any class in the system is conveniently located in one file.

The benefits of utilizing the visitor design pattern are two fold. First, the visitor pattern places all code that performs a specific function in one file, as opposed to the traditional object-oriented approach, which spreads the methods throughout the classes on which the methods operate.

The second major benefit of visitor is that once the target classes have been modified to accept visitor objects, additional new visitor classes (methods) can be added without alteration of the current classes. By eliminating the need to modify existing, proven code, the visitor pattern removes the potential that an “enhancement” will introduce anomalies through inadvertent changes to the class structure. The visitor pattern facilitates ease of maintenance, since all methods that perform a similar function are co-located. Therefore, when an enhancement is needed or a bug discovered in a method implemented using visitor, only one file needs to be modified.

One potential use of the visitor pattern is depicted in Figure 18. The information layer's semantic agent creates a semantic analysis visitor (SAV) object, then passes the SAV object in a call to the *acceptVisitor* method in the root node of the object tree under analysis. The *acceptVisitor* method in the root object utilizes the *visit* method in the SAV object parameter to call back the SAV object with itself as the parameter. The SAV object now has the root of the object tree and can perform its analysis and present its results to the semantic agent.

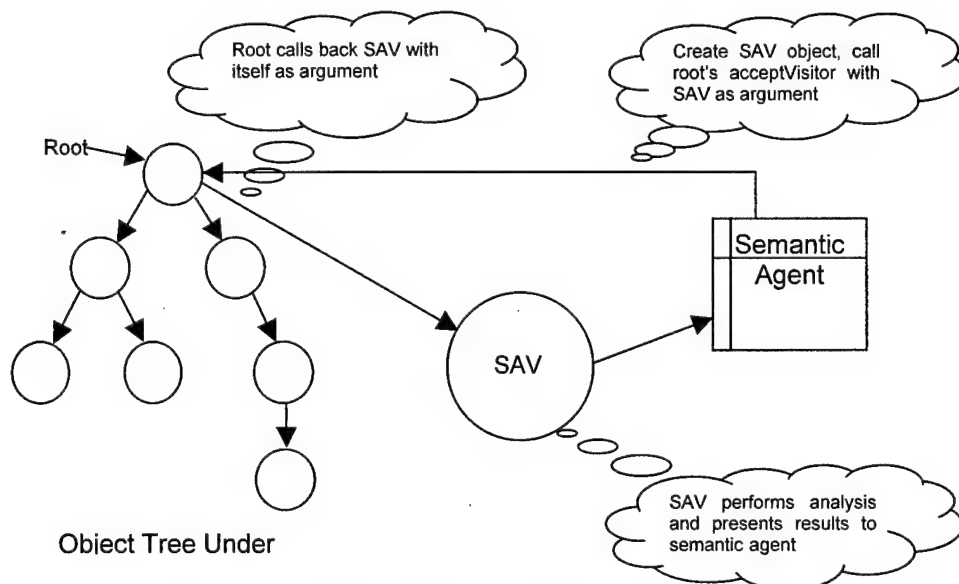


Figure 18: Use of SAV to Analyze Object Tree

In short, extending the CERTCORT class hierarchy structure to fit the visitor pattern will facilitate development of the semantic information broker, and permit additional new visitor classes to be added without alteration of the current classes. Ease of maintenance is a major benefit of the visitor pattern, since all methods that perform a similar function are co-located and existing code does not require modification. During the design and implementation portions of this effort, the possibility of employing the visitor pattern to develop portions of the semantic information broker extension to the information layer of CERTCORT will be explored.

2.7 JTree

The Java Foundation Classes (JFC) include a rich set of windowing components called *Swing* [Ste99]. The Swing classes allow graphical user interfaces (GUIs) to be developed without relying on the native windowing facilities of the operating system. *JTree* is a component of the JFC. Figure 19 [Ste99, 25] outlines *JTree* terminology and provides essential definitions for each. *JTree* provides a mechanism to present hierarchical data for display. The *JTree* component does not actually contain the data, it merely provides a view of the data. The objects that contain the data to be displayed in the *JTree* must be associated with the *JTree* object. This can be done in one of two ways.

Node: Any position within the *JTree* where data associated with the object is being represented.

Path: A collection of a contiguous set of nodes. A path can contain one or many nodes. A null path indicates a zero node path or an empty path. A collection of nodes will consist of a strict ancestry line.

Leaf: A special kind of node. As its name implies, this is the node at the end of a path.

Root: A special kind of node. In comparison to a leaf, a root's parent information is never examined. It's the highest point within the hierarchy. A root's parent information either does not exist or does not need to be displayed.

Parent: Represents a node's relationship with another node. In a parent/child relationship, the parent is analogous to a super class within the realm of object-oriented concepts.

Child: Represents a node's relationship with another node. In a parent/child relationship, the child is analogous to a subclass of its parent. It inherits all the properties associated with its parent.

User Object: Refers to the business object associated with a node. While not required, all user objects will usually be of the same class type.

Editor: A component (usually an extension of a *JComponent*) that has the unique role of allowing the user to change the data of a specific node.

Renderer: This is a component (usually an extension of a *JComponent*) that has the unique role of deciding

how a node's data is to be displayed within the context of the *JTree* when a user isn't editing the data. (Note: Using an AWT component as an editor or renderer may generate unwanted results.)

TreeModelEvents: Swing provides the following three types of events:

1. **Expansion event** – an event generated when a node is collapsed or expanded.

2. **Model events** – there are four types of model events:

a. **node changed** – generated after a node is changed. This is the only event the *TreeModel* interface supports with the method `valueForPathChanged(TreePath path, Object newValue)`. While this method could be implemented to represent any of the four types of model events, typically this represents the node changed event, and the *DefaultTreeModel* class implements it as such.

b. **node inserted** – generated when a node is inserted into the *JTree*

c. **node removed** – generated when a node is removed from the *JTree*

d. **structure changed** – a "catchall" event used when something drastic has happened to the structure of the *JTree*. It's the most expensive event as it may result in a repaint of the entire *JTree*.

3. **Selection event** – an event generated when the selection of a node takes place.

Figure 19: *JTree* Terminology

The first method involves "wrapping" the object that contains the data in a *DefaultMutableTreeNode* object. This method is sufficient if the *JTree* will only be used

to display read-only information. If the data in the JTree must be editable, this approach has several drawbacks [Ste99, 26-27] as outlined below.

- It demands that the application constructing the JTree take full responsibility for constructing and maintaining all the hierarchical relationships between each node.
- The responsibility of keeping concurrent data accurate falls back on the application containing the JTree.
- The *DefaultMutableTreeNode* is not a thread-safe class.

The second, and preferred, method associating the data object to the JTree is to implement the *MutableTreeNode* interface. Implementing the interface provides a “bridge” between the user object class being displayed and the Swing *MutableTreeNode* interface [Ste99, 27]. This bridge provides a means of translating API calls invoked in the JTree to corresponding methods in the user’s object. Implementing the interface eliminates any requirement for the user object class to know the functionality of the JTree and vice versa.

The JTree component of Swing in the JFC provides the ability to view hierarchical data in an “outliner-style” tree. The most flexible means of utilizing the functionality of a JTree is through implementation of its *MutableTreeNode* interface. Use of the JTree’s capabilities will be further explored in development of the semantic broker’s user interface.

Since any approach to enhancing the semantics of simulation components will involve the application, or at a minimum the understanding of, multi-agent systems, this topic is discussed next.

2.8 Agent Technology

An *agent* is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives

[Wei99, 29]. An agent performs actions on or in its environment and monitors the results of its actions. It uses this feedback to determine if the desired state has been achieved or if further action is required. Although an agent normally cannot exercise total control over its environment, most agents have some influence over a portion of their environment. The collection of actions an agent has available to modify its environment is known as its *effectoric* capability. Since most real world environments are non-deterministic, the same action may leave the environment in different states depending on whether the pre-conditions of the action were satisfied. This fact makes it essential that agents be capable of deciding which action to perform and of dealing with the failure of an attempted action. There are five environmental properties that affect the complexity of the decision-making process [Wei99].

- **Accessible vs. Inaccessible:** an environment is accessible if an agent can obtain complete, accurate, up-to-date information about the environment's state.
- **Deterministic vs. Non-Deterministic:** an environment is deterministic if an action has one guaranteed affect.
- **Episodic vs. Non-Episodic:** in an episodic environment, the performance of an agent is dependent on a number of discrete episodes, with no link between its performance in different scenarios.
- **Static vs. Dynamic:** in a static environment, the agent can assume only its actions change the environment's state. In a dynamic environment there are other processes, over which the agent has no control, affecting the agent's environment.
- **Discrete vs. Continuous:** an environment is discrete if there are a fixed, finite number of actions and percepts in it.

The most complex environments are those that are inaccessible, non-deterministic, non-episodic, dynamic, and continuous [Wei99].

2.8.1 Intelligent Agents

The previous definition of an agent would include a simple thermostat, since such a device is capable of monitoring its environment and performing actions to modify it. To

extend an agent into the realm of intelligent agents, it must be capable of flexible autonomous action to meet its objectives. This flexibility embodies three things [Wei99]:

- **Reactivity:** able to perceive their environment and respond in a timely fashion;
- **Pro-Activeness:** able to exhibit goal-directed behavior by taking the initiative;
- **Social Ability:** capable of interacting with other agents (and possibly humans).

Intelligent agents must verify that the pre-conditions of an action are satisfied before executing the action. Also, an intelligent agent must decide what to do in the event another process changes the state of the environment, and nullifies the pre-conditions, while the action is being performed. Usually this results in a failure of the action, so the agent must determine another course of action that achieves its design goals.

2.8.2 Agents and Objects

At first glance, agents seem very similar to objects. After all, objects encapsulate their data and provide methods that access this data. Therefore, objects seem to have autonomy over their state. However, the public methods contained in the object are executed by external procedures. It has no control over when or if these methods are executed, and it has no ability to decide whether it is in its best interest to execute the method. An object has no control over its behavior.

By contrast, an agent receives a request to perform a specific action. The agent decides whether accomplishing the action will help it achieve its goals. If so, the agent complies with the request. However, control lies with the agent, and the agent controls its behavior.

2.8.3 Agent Architectures

The framework within which an agent senses its environment and executes actions to influence that environment is the agent's architecture. In his book [Wei99], Gerhard

Weiss considers four architectural classes of agents. These are logic based, reactive, belief-desire-intention, and layered architectures.

Logic based agents act as theorem provers in a framework where both its desired behavior and the environment's state are represented symbolically. Agents use the rules of formal logic to deduce which actions will lead to goal satisfaction.

Reactive architectures are based on the concept that intelligent, rational behavior is inseparably linked to the environment in which the agent operates, and intelligent behavior is the aggregation of simpler agent-environment interactions. In reactive architectures, agents sense their environment and map perceptual input directly to actions. Formally, this might be written as *situation* \rightarrow *action*. To deal with the possibility a particular perceptual input maps to more than one action, many reactive agents employ layers to determine which action will be performed. Under this architectural scheme, lower layer actions inhibit higher layer actions. This allows high priority actions to be placed in the lower layers where they will have execution priority over lower priority actions in the higher layers.

Belief-Desire-Intention (BDI) agent architectures attempt to give agents the ability to understand practical reasoning. In this framework, an agent develops an intention based on a set of available options. This intention drives future means-ends reasoning, constrains future decision making processes, and persists until environmental changes make the intended goal unachievable.

Layered architectures decompose an agent into different layers, each of which deals with a different type of behavior. In general terms, there are two types of layering: horizontal and vertical.

In a horizontally layered architecture, each layer is directly connected to the sensory input and action output. These architectures normally include a mediator function that

determines which layer has control. This mediator eliminates conflicts when two layers simultaneously detect an environmental change and generate actions. However, designers must construct the mediator so it knows how to resolve all possible conflicts between layers. In a system with n layers, each of which can suggest m actions, there are m^n possible agent interactions to be considered [Wei99, 62]. Defining rules for resolving each of the possible conflicts dramatically increases the complexity of the design and introduces a system bottleneck.

Vertically layered architectures require all perceptual input to travel up to the highest level before an action is determined. In the vertically layered architecture, all input is sensed by exactly one layer, and all actions are performed by exactly one layer. The input layer is always the lowest, or bottom, layer. The layer that performs actions depends on whether the architecture is one-pass or two-pass. In a one pass vertically layered architecture, input enters the bottom layer and actions are executed by the top layer. A two-pass vertically layered architecture requires that input enter the bottom layer and pass upward through all layers above. Once at the top layer, that layer determines an action and passes it down to the next lowest level. Each successively lower layer processes the action, adds additional actions as appropriate, then passes it to the next lower level. When the action reaches the lowest layer, it is executed. This architecture reduces the number of possible layer interactions to be considered significantly; but reduces the fault tolerance of the system, since control must pass to each layer before a decision can be made.

2.8.4 Multi-Agent Systems

To accomplish real-world goals, most agent-based systems employ many agents working together to accomplish the desired objectives. These systems provide

frameworks for agent communication and interaction. Agents communicate via message passing. The interaction protocol defines which messages may be sent in response to a received message as well as those that may be sent to initiate interactions. In this respect, an interaction protocol governs an exchange of a series of messages called a *conversation*.

Coordination of agent activities is essential in a distributed, multi-agent system. Coordination can be divided into two categories depending on whether the agents are non-antagonistic or competitive. Cooperation among the former is coordination, while in the latter case it is termed negotiation [Wei99, 83]. In a system where some agents compete and others coordinate their efforts, each agent must maintain a model of other agents in its environment and update the model as new agents enter the environment, goals change, etc.

The contract net protocol [Wei99, 100-101] is a widely used protocol for distributed tasks. Under this protocol, an agent wanting a task performed is called the *manager*, and agents able to perform the task are called *contractors*. The manager agent announces the task to be performed, receives and evaluates bids from contractor agents, awards a contract to a suitable contractor, and receives and synthesizes the results. The contractor agents receive the task announcements, evaluate their capability to perform a specific task, decline or bid on the task, perform the task if their bid was accepted, and report their results. In the contract net protocol, agent roles are not predetermined. This allows an agent that previously acted as a contractor to bid on a task, to break that task into sub-tasks, and, acting as a manager, announce several of those sub-tasks open for bid. The resulting manager-contractor links form a control hierarchy for task sharing and result synthesis [Wei99, 101].

The multi-agent CERTCORT layered framework manipulates data stored as objects. To provide persistence for these objects, an object-oriented database management system is utilized. The capabilities and vulnerabilities of these systems is discussed next.

2.9 Object-Oriented Database Management Systems

An object-oriented database management system (OODBMS) provides object persistence and the consistency of atomic transactions. These systems free application designers and programmers from the task of developing and implementing a persistency scheme for each application developed.

2.9.1 Object-Oriented Database System Manifesto

When Malcolm Atkinson and company [Atk89] wrote "The Object-Oriented Database System Manifesto" in 1989, they outlined the main features and characteristics a system must have to qualify as an OODBMS. These features include:

- **Complex Objects:** objects composed of other objects. The *manifesto* lists the set, list, and tuple as the minimum set of constructors.
- **Object Identity:** can be existence based or value based. A system that maintains object identity based on a unique attribute value places the burden of maintaining uniqueness of object identifiers and referential integrity on the user. In a system that supports existence based object identity, the system ensures uniqueness of identifiers and maintains referential integrity.
- **Encapsulation:** the data structure and methods that manipulate it are wrapped in an interface. The only means of accessing the data structure is through the methods defined in the interface. Encapsulation provides some level of logical data independence, since the underlying implementation of an object can be changed without changing the interface and the applications that use the data.
- **Class:** a template for creating an object. A class contains two aspects: an object factory and an object warehouse [Atk89, 7]. The object factory is used to create new objects of the class, and the object warehouse is the collection of all objects that are instances of the class.
- **Inheritance:** allows the extension of a general class into one or several more specialized classes. Additionally, inheritance helps in factoring out shared specifications and implementations in applications [Atk89, 8].

- **Late Binding:** allows the same operation name to be used for multiple, different classes of objects. Late binding means that method parameters are bound to object class types at runtime. The runtime system determines which method to call based on the data type and number of parameters.
- **Extensibility:** means that the system comes with a set of predefined types and new types can be added. When new types are added they can be used in the same ways as predefined types.
- **Persistence:** means that objects are stored when an application terminates and can be retrieved and loaded the next time the application is started.
- **Secondary Storage Management:** the set of mechanisms required to manage large databases. These mechanisms include index management, data clustering, data buffering, access path selection, and query optimization [Atk89, 12].
- **Concurrency:** the ability of the system to allow multiple users to access the system.
- **Recovery:** the ability of the system to recover from hardware and software faults.
- **Ad Hoc Query Facility:** provides the user with the ability to express non-trivial queries concisely and is application independent.

Since the *manifesto* was written, several OODBMS packages have entered the mainstream. While still not as popular as relational systems, they are gaining ground in some areas. The OODBMS utilized in CERTCORT is ObjectStore, and, as such, it will be the focus of the remainder of this OODBMS discussion.

2.9.2 ObjectStore

ObjectStore provides native support for storing objects. The term *native* meaning no conversion of the object (i.e. object-relational mapping) is required to make the object persistent. ObjectStore uses a postprocessor on the Java class file to add the additional code to make the object persistent. However, the security mechanisms in Java prohibit changes to built-in classes. Therefore, container classes like *Vector* and *Hashtable* cannot be annotated by the compiler. One solution to this problem is to substitute “work-alike” versions of these classes provided with ObjectStore. This approach requires some code modification, but the changes are relatively minor. Two other issues that

should be addressed are the requirement for transactions, and the need to specify whether objects should be retained after a transaction. First, all manipulation of persistent objects must be done within the confines of a transaction. Any attempt to modify objects outside transaction boundaries causes an exception. The second issue deals with what happens to the contents of an object after transaction completion. If the user does not specify a retainment level, after transaction completion the objects are *hollow*. This means the shell of the object is there, but any attempt to access attributes will result in an exception.

The CERTCORT system utilizes ObjectStore to achieve object persistence, but the source databases for SUPPRESSOR, MSFD, etc. are essentially heterogeneous flat files from various sources. This fact makes a review of multi-database systems essential, and this topic will be covered next.

2.10 Multi-Database Systems

Multi-database systems are composed of separate, heterogeneous, autonomous data sources. The heterogeneity may manifest itself in the structure of the database or the Database Management System (DBMS) in use. These systems are autonomous because, quite often, the various local databases are not under the control of a single person or organization. One reason is the case of several organizations sharing portions of their data in order to facilitate a strategic partnership. These organizations need to share data to gain a competitive advantage, but at the same time cannot give up control of their information resources. Several issues arise in the attempt to resolve differences among heterogeneous data sources.

- **Schema Differences:** These can be eliminated by developing specialized procedures to retrieve data from each unique source.

- **Identical data item, different names:** This problem can be overcome by making each source's attribute name an alias and referencing a common standard data element.
- **Different Units:** Even if two data elements are identically named and have the same overall meaning, differences in units of measure must be rectified in order to share data among the various sources.

To make interoperability of heterogeneous sources transparent to the users of the system, these syntactic and semantic differences must be overcome. One approach to resolution of this problem is to develop and maintain a data dictionary and data encyclopedia as were discussed in section 2.4.1.

Developing a global schema and designing methods that map the elements of the heterogeneous sources into the unified structure is another approach to solving this problem. This method employs schema integration techniques to develop the global schema, and runtime routines to populate the schema from the various data sources. Ashby's thesis [Ash00] work focused on applying formal methods and knowledge-based engineering techniques to develop a transformation system that integrates heterogeneous data sources into a common object model. Colonese [Col99] also focused her efforts on developing a common object model for heterogeneous sources; however, her work focused on utilizing manual schema integration techniques and an *Integration Dictionary* that provides semantic interoperability among the sources.

The CERTCORT data sources, in some respect, fall into the category of federated databases. Each source is created by a different organization and has a different structure. However, the way CERTCORT source data is used differs dramatically from most conventional databases. The ultimate goal of CERTCORT is to allow reusability of scenario components across simulator platforms. Each source database has its own unique syntax and the representation of scenario components varies substantially from one simulation system to the next. These factors make the development of a global

schema for CERTCORT a massively complex problem. Additionally, translators must be developed to convert, for example, a scenario component from SUPPRESSOR to the Joint Interim Mission Model (JIMM). This functionality will permit users to create new scenarios from components of several different scenarios in several different formats.

2.11 Summary

This chapter presented a review of the various technologies that form the cornerstone of this research. First, the structure of the SUPPRESSOR scenario files and the various input sources to the creation of these files was studied. The CERTCORT class hierarchy was reviewed to determine appropriate extensions to enhance semantics of the model. The role of metadata, what it is and how it can be exploited to extract meaning from a scenario component was examined. The visitor design pattern was explored to uncover its capabilities and potential use in development of the semantic broker's analysis engine. The Java Foundation Class component JTree was scrutinized to discover its capabilities and nuances. Agent technology was researched to examine the portions of that technology applicable to this research. Object-oriented databases, and, in particular, ObjectStore were explored to understand their capabilities and limitations. Finally, the constraints and issues specific to heterogeneous Multi-Database Systems were the final topic of this literature review. In the next chapter, a general methodology is developed to apply these technologies to extending the semantic representation capabilities of CERTCORT.

3. METHODOLOGY

3.1 Introduction

This chapter defines the methodology used to develop a semantic broker capable of providing users with effective scenario component retrieval and transformation. It begins with a brief discussion of the tools used to perform the analysis, design, and implementation of the semantic broker. Following the discourse on tools is a discussion of the general approach utilized in the design of the semantic broker. Here, the subjects of scenario component representation (i.e. the object model), component generation, relevant component retrieval, and component transformation are discussed. Finally, the design of the semantic broker is discussed in detail.

3.2 Development Tools

The development of the semantic broker agent utilizes object-oriented tools and techniques. This section identifies those tools and provides some insight into their capabilities and the benefits of their use.

3.2.1 Object Modeling

Object-oriented development encompasses the analysis, design, implementation, and maintenance of software systems using layers of abstraction to model the real-world system. The object-oriented approach has the following three advantages [Mul97, 15]:

- The stability of models with respect to real-world entities.
- Iterative construction, which is made easier by the weak coupling between components.
- The ability to reuse elements across development projects.

Object-oriented development utilizes a modeling language during the analysis and design phases to accurately capture the entities and relationships in the real-world system being modeled.

3.2.1.1 *Unified Modeling Language*

In the early 1990's, James Rumbaugh, Grady Booch, and Ivar Jacobson were developing three separate object-oriented methodologies. These were *Object Modeling Technique (OMT)*, *Booch`93*, and *Use Cases*, respectively. As the differences between the three techniques began to dwindle, these three pioneers determined to collaborate and consolidated their methods into the *Unified Method*. The *Unified Method* has further evolved and is now known as the *Unified Modeling Language (UML)*.

UML provides the notation necessary to describe the elements and associations of a problem and the tools required to express the selected solution to the problem. These two activities are known as *analysis* and *design*, respectively [Mul97, 11]. UML defines six different types of models and nine different types of diagrams. Of these, only the class diagram is used extensively in this effort. The class diagram shows the static structure of the system—its entities and their associations. UML is used throughout this work for the purpose of documenting problem areas and solutions, and is one of the three modeling languages supported by the modeling environment selected for this research.

3.2.1.2 *Rational Rose*

The Rational Rose (Rose) object modeling environment is produced by Rational Software Corporation, Santa Clara, California. The Rose modeling tool is used throughout this research to develop the UML diagrams. Rose provides a point-and-click

modeling environment and has limited code generation facilities. Rose is capable of generating Java or C++ shells, depending on the selected implementation language.

3.2.2 Programming Language

The Java programming language is used as the implementation language for this research. Java was developed by Sun Microsystems to be simple, reliable, and architecture neutral. This programming language has many desirable features including [Far98, 6-21]:

- **Object-Oriented Environment:** Java is a pure object-oriented programming language. A data structure or function cannot exist or be accessed at runtime except as an element of a function.
- **Abstract Interfaces:** An interface describes the operations, messages, and queries a class of objects is capable of servicing without providing any information about how these operations are implemented. This feature allows implementation-neutral interfaces to be specified for a system.
- **Platform Independence:** Java source code is compiled into bytecodes and can be executed on any Java Virtual Machine (JVM) regardless of the JVM's underlying hardware and operating system.
- **Exception Handling:** Java supports throwing and catching exceptions, both system- and application-defined.
- **Network Support:** Java includes multilevel support for network communications. Low-level sockets can be established between processes and data communication protocols can be layered over the socket connection.
- **Security:** Java provides both a secure local environment and the ability to engage in secure remote transactions.
- **Multithreading Support:** In Java, any class can extend the *java.lang.Thread* class by providing its own implementation of a *run()* method. When started, this object will execute in a separate thread.

These features combine to produce a programming language that is robust, better facilitates implementation of object-oriented designs, and is truly platform independent. At the time of this research, Sun Microsystems' most recent version of the Java language is the Java Development Kit 1.3 (JDK 1.3). This release provides

enhancements with regard to execution times for certain operations. Sun's JDK 1.3 is utilized in this research for implementation of the semantic broker agent.

3.3 General Approach

This section provides a description of the general approach selected to develop the semantic broker in this research. First the object model used to represent scenario components is presented, followed by a discussion of the semantic broker's functions.

3.3.1 Scenario Component Representation

To facilitate the addition of new simulator types without necessitating major changes to the scenario component model, the semantic broker utilizes a generic or common model for scenario component representation. This model is shown in Figure 20.

Utilizing this object model for scenario components allows the representation of virtually any simulator system's scenario components. However, this method requires that the details of each component type be provided through metadata. As the figure shows, a *MetaComponent* consists of *SComponents*, and each *SComponent* is derived from a *MetaSyntaxUnit*. Additionally, each *SComponent* object can have one or more child *SComponent* objects.

In this scheme, a *MetaComponent* object represents the semantics of a scenario component. For example, a *MetaComponent* object named *SAM Missile* is comprised of the components that make up a surface-to-air missile in a simulation scenario. The *MetaComponent* class contains semantic information about the scenario component. Ideally, a *MetaComponent* object would have a name attribute value that identifies the real-world object represented by the scenario component. Additionally, it would also contain information in its *comments* attribute describing the capabilities and limitations of the scenario component.

Objects of the *MetaSyntaxUnit* class represent the syntax definition of a specific scenario component (i.e. PLAYER-STRUCTURE, TACTIC, COMM-RCVR, etc.) of a specific type of scenario source (i.e. SUPPRESSOR, SWEG, JIMM, etc.). During scenario source file parsing, the file parser references these objects to determine how to proceed with parsing of the component definition.

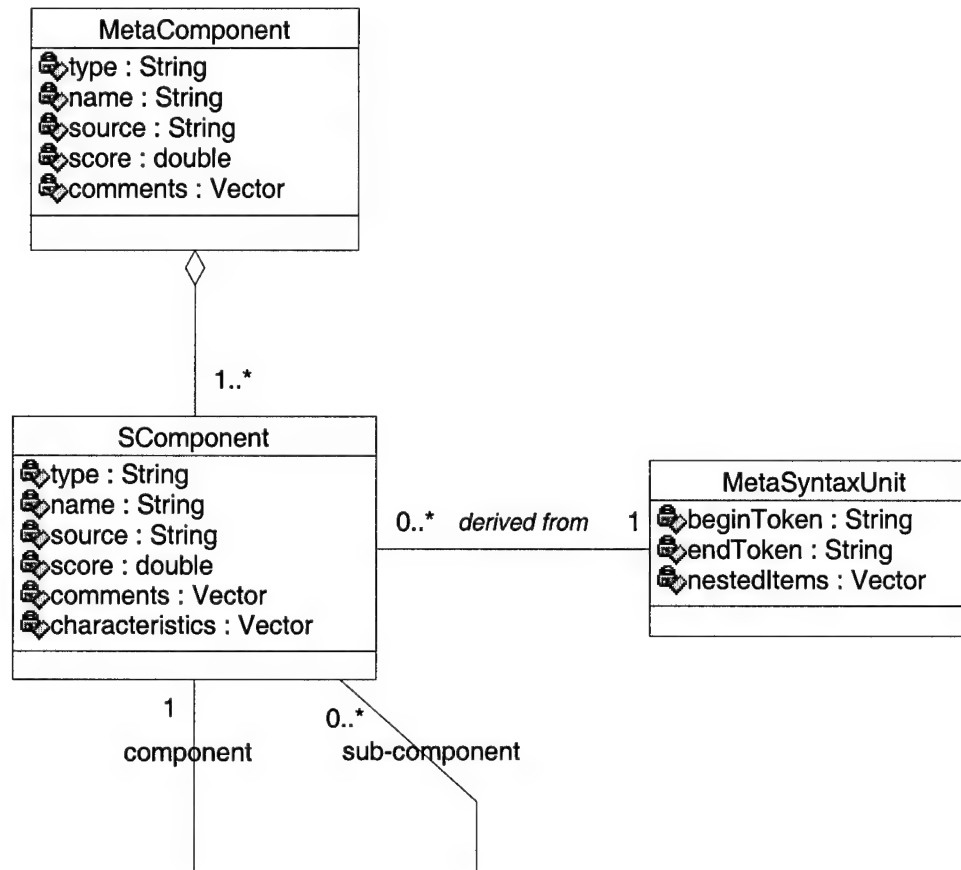


Figure 20: Scenario Component Representation

In addition to providing semantic representations of entire scenario components, (e.g. a PLAYER-STRUCTURE in SUPPRESSOR) the semantic broker must also be able to provide semantic interpretations of portions of scenario components. For example, an analyst may require a communication receiver. It is unlikely that such a system exists as a high-level component like a PLAYER-STRUCTURE, but in all

probability the required scenario component does exist as a sub-component of a PLAYER-STRUCTURE. The component representation shown in Figure 20 facilitates representation of these sub-components as well, since it does not differentiate between components and their sub-components except through the parent-child association.

Figure 21 provides an example of how a portion of two SUPPRESSOR PLAYER-STRUCTURE scenario components are represented using this scenario component representation. During component generation (the process of creating object representations from source file text definitions), the parser creates two containers of *MetaComponent* objects. The first, called *metaComponents*, contains links to the high-level scenario components (in this case PLAYER-STRUCTURE components). The second container, called *metaSubComponents* contains links to the high-level component's sub-components (in this case TACTIC components). The extra container for the sub-components is a specially created index used to avoid lengthy traversals when performing searches for sub-components.

The scenario component representation scheme shown in Figures 20 and 21 provides extensibility for adding new types of simulation system data sources. Adding a new type requires providing a *MetaSyntax* file that contains syntax definitions for all the components of the new system's scenario source file format, and providing a parser that is capable of generating components from the new scenario type's source files.

The scenario component object model presented here is extremely simple. Under the scheme presented, the object tree representation of a scenario component is composed entirely of *SComponent* objects. This, of course, excludes the *MetaComponent* root object, which merely exists to enhance the semantics of the component. This object model provides the greatest flexibility in representing different simulation type's scenario components. Under this scheme, virtually any type's scenario

components can be accurately represented. The simplicity of the object model also affects the complexity of the component generation process.

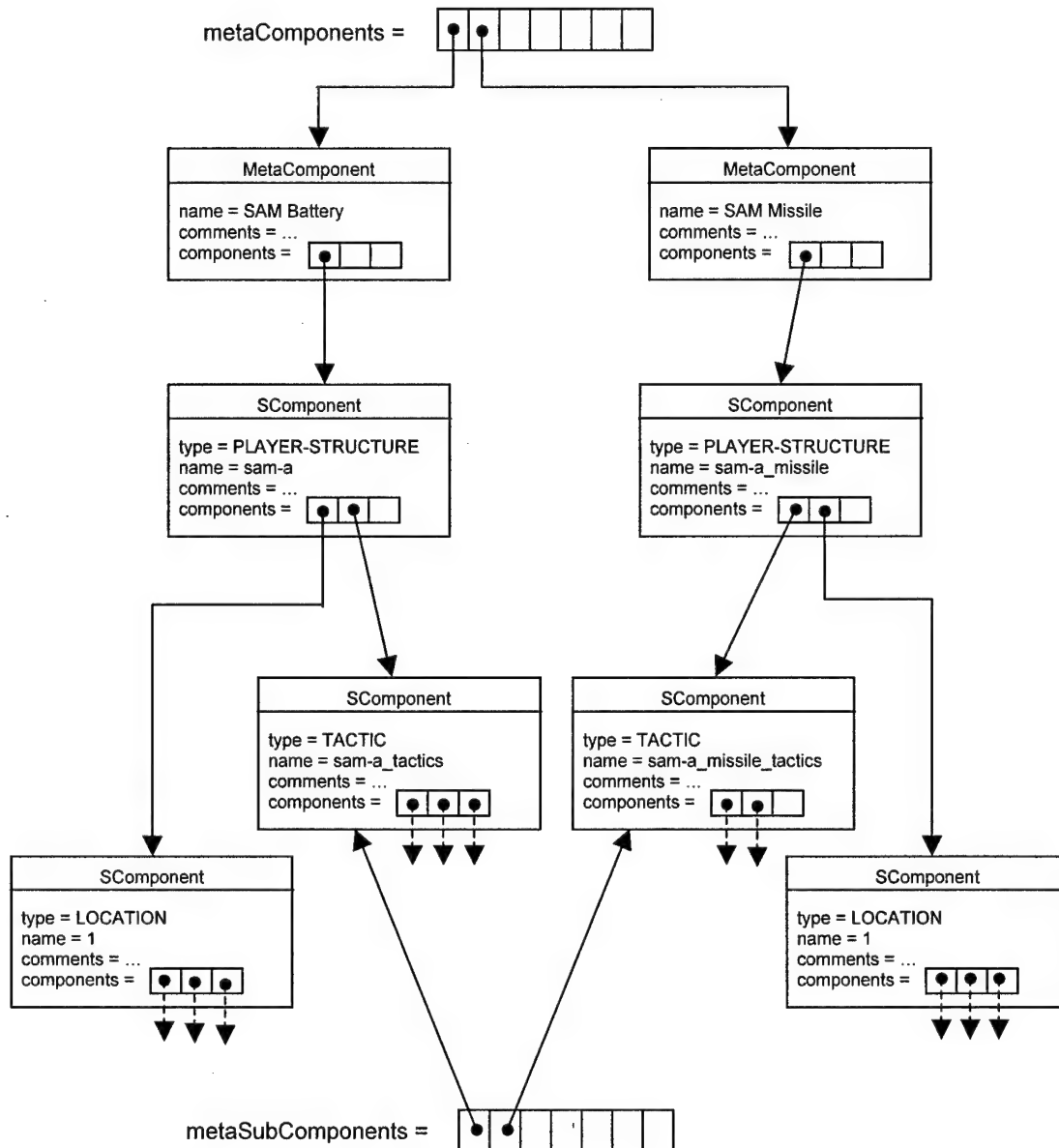


Figure 21: Sample Scenario Component Representation

3.3.2 Component Generation

To accomplish the task of generating scenario components from the text-based files of the scenario source database, a text parser is required that can translate the

components from their textual definitions to *SComponent* object trees. In order to facilitate the analysis of scenario source components, both source and *signature* components must be generated by the semantic broker. The classes and data sources involved in the component generation process are depicted in Figure 22.

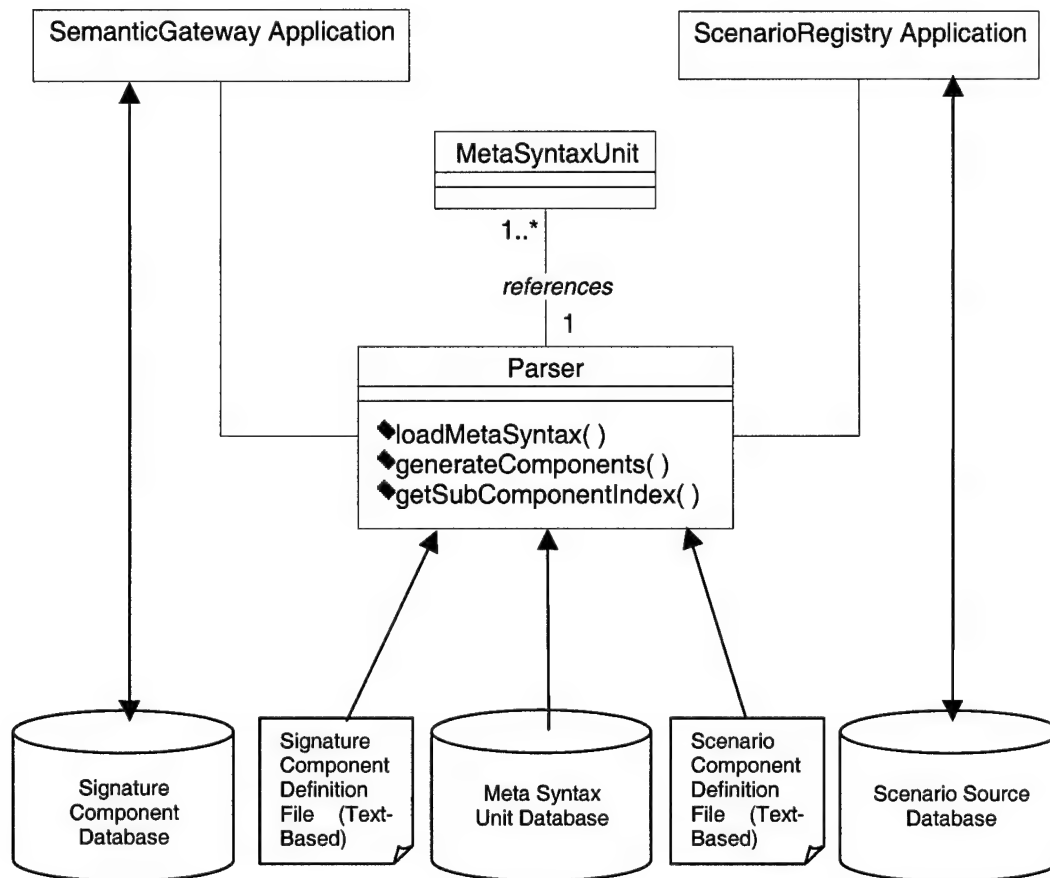


Figure 22: SemanticGateway Component Generation

Both the *SemanticGateway* application and the *ScenarioRegistry* application utilize parsers to generate *SComponent* object tree representations of scenario components. The *SemanticGateway* application uses parsers to generate new signature components for the *Signature Component Database* (SCDB). The SCDB is discussed in Section 3.3.3.1 along with signature analysis. For the purposes of this discussion, it is sufficient to understand that signature components are created from text files containing signature

definitions. These files are very similar in format to scenario source files. Both signature and scenario components are generated from their text-based definition only once, then they are stored in their object format in the SCDB and *Scenario Source Database*, respectively.

The *ScenarioRegistry* application utilizes parsers to generate components from a scenario source definition file when it is registered with the system. These source components are stored in the Scenario Source Database and are accessed and compared to the query's signature object during the relevant component retrieval process. This topic is discussed in detail in Section 3.3.3.

The *generateComponents* method of the *Parser* class parses the scenario source file and returns a list of scenario components. Actually, the list contains the root *SComponent* objects of the object tree representation of each scenario component definition contained in the source file. A call to the *Parser* object's *getSubComponentIndex* returns a list of references to the sub-components of each scenario component. Essentially, this list is a flattened hierarchy and eliminates the need to perform an exhaustive search of each object tree when looking for a specific sub-component. This separate index is shown in Figure 21 of Section 3.3.1.

There is a parser for each type of scenario source files the system recognizes. Each of these parsers must extend the *Parser* class. *Parser* objects create and maintain a list of *MetaSyntaxUnits* for the type of source file being parsed. There is a *MetaSyntaxUnit* object for each type of component that can be extracted from a scenario. These objects contain the start and stop tokens for a particular component and the types of nested items. *Parser* objects create *MetaSyntaxUnit* objects from syntax definitions found in the *MetaSyntaxUnit Database* (MSUDB). The syntax for *MetaSyntaxUnit* object definitions is:

START-TOKEN [attribute*] [component | componentRef]* **END-TOKEN**

This definition requires that a component definition begin with a valid START-TOKEN followed by optional attributes, followed by either one or more *component* or *componentRef* tokens (also optional), followed by an END-TOKEN; which may be the keyword *NULL* if the component has no stop label.

This definition requires an explanation of the difference between a *component* and a *componentRef* in the syntax definition. If a *component* token appears in the definition, the child component's definition is nested inside the parent component's definition. If the *componentRef* token appears, only a component type and identifier are nested inside the parent's definition. In the latter case, the actual definition of the child component is located elsewhere in the scenario file representation, and must be linked with its parent after all components have been generated. Figure 23 provides a sample of *MetaSyntaxUnit* definitions for SUPPRESSOR data sources.

Use of the MSUDB provides some level of flexibility in determining level of detail at which components are generated. For example, as defined in Figure 23, *ZONE-CHARACTERISTICS* components have attributes but no nested child components. This means that all items between the start and end tokens are treated merely as characteristics of the parent component, and, as such, cannot exist on their own. Since these characteristics are not components, they have no semantics. However, changing the definition of this component to include child components, and, of course, adding the definitions for those child components, allows the level of detail of the *ZONE-CHARACTERISTICS* component to be increased. It is important to note that this change in level of detail is achieved without modification of source code.

Extensibility of the component generation portion of the semantic broker architecture is achieved in the following manner. When a new simulator type is added to the semantic broker framework, the following software and data sources must be provided:

- Signature and source component generators that extend the abstract *Parser* class.
- Syntax definitions for the new system's components must be added to the MSUDB.
- Definitions of prototypical components of the new system must be added to the Signature Component Database.

Adding the above software components facilitates component generation for the new system and sets the stage for relevant component retrieval.

```
PLAYER-STRUCTURE attribute component END PLAYER-STRUCTURE
TACTIC component END TACTIC
CAPABILITY component END CAPABILITY
LINKAGES attribute NULL
SUSCEPTIBILITY component END SUSCEPTIBILITY
ASG-CMD-CHAIN attribute NULL
EVALUATION-RATES attribute END EVALUATION-RATES
INTELL-REPORT-FREQ attribute END INTELL-REPORT-FREQ
MAX-MSG-ATTEMPTS attribute NULL
MAX-SNR-PERCEPTIONS attribute NULL
MOVE-TO-ENG attribute NULL
MSG-RPT-GUIDE attribute END MSG-RPT-GUIDE
SALVO-FIRING attribute END SALVO-FIRING
SNR-RPT-GUIDE attribute END SNR-RPT-GUIDE
ZONE-CHARACTERISTICS attribute END ZONE-CHARACTERISTICS
THINKER componentRef NULL
SNR-RCVR componentRef NULL
```

Figure 23: Sample MetaSyntaxUnit Definitions

3.3.3 Relevant Component Retrieval

To retrieve relevant scenario components from the myriad of source files available, users must specify their requirements in a query. Traditional query languages, such as the Standard Query Language (SQL), are not useful here, since the object representations of scenario components are not standardized in size or complexity. Additionally, a search for even a simple scenario component would require an extremely

complex SQL query, if, in fact, SQL could be used at all. To deal with this problem, this research utilizes signature analysis to identify relevant components.

3.3.3.1 Signature Analysis

To facilitate the development of a semantic broker capable of retrieving relevant scenario components, this research utilizes a database of signature components, which the semantic broker uses to analyze the contents of existing scenario components. A high-level abstraction of this signature analysis approach is shown in Figure 24.

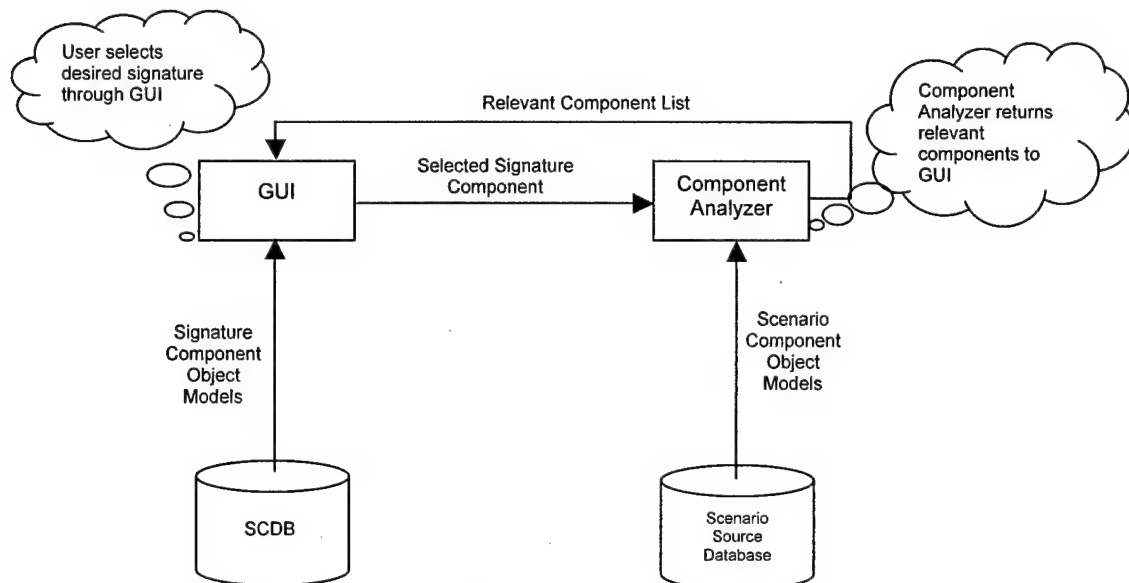


Figure 24: Signature Analysis Approach

In the figure, the *Component Analyzer* utilizes a signature component to determine which components in the Scenario Source Database to include in the relevant components list returned to the GUI. Signature components are essentially generic definitions of semantic entities. For example, the signature component for an air interceptor defines the essential sub-components and characteristics a scenario component must have to be interpreted as an air interceptor. The signature is a baseline entity, and any component that contains it as a subset will be interpreted as relevant. Each signature component has a semantic tag that identifies its contents (e.g.

Air Interceptor, Bomber, etc.), and comments that describe the capabilities of the signature.

Returning now to Figure 24, the list of signature components is retrieved from the SCDB into the Graphical User Interface (GUI) where it is presented to the user in the form of a scrollable list. The scrollable list contains the semantic tags associated with each of the signature components. The user selects the desired signature component from the list. The selected signature component is then sent to the *Component Analyzer* where it is used to search for existing scenario components matching its composition. The output of the *Component Analyzer* is a list of existing scenario source components deemed relevant by its analysis process. Furthermore, this list of components is sorted based on the relevance score assigned to each component by the *Component Analyzer*. The *Component Analyzer* scores a scenario component on how well its sub-components and characteristics match the signature component's sub-components and characteristics. The closer an existing component matches the signature component, the higher that component's relevance score. The relevance score ranges from zero to one. A scenario component that scores a one contains, within its object tree structure, an exact replica of the signature component. The relevance score assigned by the *Component Analyzer* is largely determined by the semantic representation's level of abstraction.

3.3.3.2 *Level of Abstraction*

In designing a semantic broker, a key design decision is the level of abstraction involved in representing the semantics of scenario components. Figure 25 provides a graphical depiction of the design tradeoff. The more abstract the representation, the more likely the system will overwhelm the user with too many "relevant" components.

Conversely, the more detailed the representation, the more likely the system will miss relevant components or return an empty list as a result.

The obvious solution to this problem is a compromise between very abstract and very detailed representations. Therefore, signature components are as generic as possible. From this generic baseline a user may increase the level of detail and thereby decrease the number of perceived relevant components returned by the *Component Analyzer*.

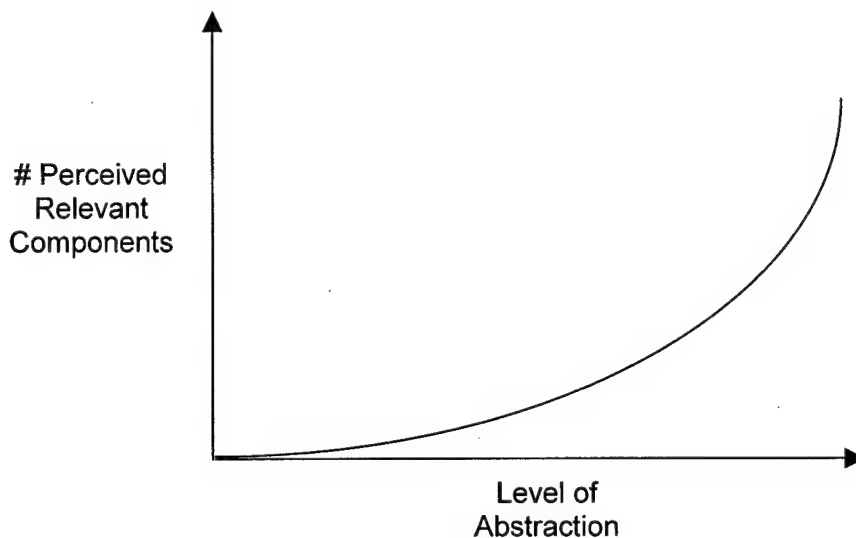


Figure 25: Level of Abstraction Tradeoff

The relevant component retrieval process presented here uses signature components as complex query structures. The user selects a generic signature component (e.g. Air Interceptor, Bomber, etc.), modifies its attribute values and sub-components to match the desired search criteria, and initiates a search. Relevant scenario source components are scored based on how closely their structure and characteristics match those of the signature component. Once a list of relevant components has been produced, the user selects the most suitable one for inclusion in

the new scenario. If the selected relevant component's type (i.e. SUPPRESSOR, SWEG, etc.) is different than that of the scenario being constructed, the component will have to be transformed to the target format.

3.3.4 Component Transformation

Component transformation is the process of translating a scenario component developed to execute in one simulator system to conform to the syntax of another simulator's format. In situations where the translation is too complex to be handled by the automated process, or the scenario item does not have a comparable counterpart in the target scenario format, techniques must be developed to effectively represent the nontransferable data in a manner that allows the human analyst to deal with the problem.

Transformation of components from one simulator format to another is perhaps the most difficult facet of scenario reuse. A successful transformation technique must effectively deal with all the translation categories outlined in Figure 26 [LSA98]. It should be noted here that, in the figure, the original author's use of the phrase *data item* has been replaced with *component* to conform to the terminology of this report. The categories of Figure 26 run the gamut from components that are identical, and no translation is required, to situations where the component in Model-A can not be represented in Model-B.

The first conversion category in Figure 26 is self-explanatory. At first glance, the second category seems identical to the first. However, the second category differs in the following way: In the first, the component in Model-A is identical to that in Model-B (i.e. Data Item Model-A = Data Item Model-B). In the second category, all the keywords present in the component of Model-A are present in that of Model-B; however, the

component of Model-B may, in fact, have more keywords that are not present in the component of Model-A (i.e. Data Item Model-A \subseteq Data Item Model-B). In the case where the component of Model-B has more keywords than that of Model-A, the values of those additional keywords must be set to some predetermined innocuous value.

Conversion Category	Definition	Transformation
Compatible	Component in both models with no keyword differences.	None.
Fully Upward Compatible	Component in both models with all Model-A keywords in Model-B.	None.
Upward Translatable	Component in both models, some keyword differences, but no functional differences.	Either 1) add the Model-A keyword as a synonym or 2) translate the Model-A keywords into Model-B keywords.
Convertible	Component in both models, both keyword and functional differences, including components where the ordering has changed.	Logic is built into the transformation program that converts Model-A to Model-B, with each convertible data item having a module dealing explicitly with it.
Replaceable	Component in Model-A only, but functionality is represented in one or more Model-B data items.	Automation may be too difficult and will require the intervention of a user to manually adjust.
Non-Replaceable	Component in Model-A only and functionality not represented in Model-B.	Traceability in Model-B via commented-out data item blocks may be appropriate, along with comments indicating the reason why they cannot be put into Model-B format.

Figure 26: Translation Categories of Data Items

The third category in Figure 26 deals with the situation where the component exists in both Model-A and Model-B; however, some of the keywords have exactly the same meaning, but different names. In this case of synonymous keywords, their relationship can be documented in metadata and referenced during transformation. This alleviates

the need to hard code every synonym relationship in the code and increases the flexibility of the system.

The last three categories of transformations exemplify the difficulties encountered in scenario reuse. The fourth category illustrates a situation in which it may be impossible to avoid hard-coding the transformation process in the source code. The fifth transformation category deals with transformations where the functionality of a component in Model-A is split-up among several components in Model-B. In some cases, it may be possible to hard-code these transformations in the source code; however, in others the process may be so complex that it requires a human analyst's intervention. The last category of transformations, while easier to deal with than those of the previous category, has the most serious effect on the new scenario. Here the untransformed component from Model-A can be placed in Model-B and commented out; however, the functionality of that component is completely lost and will have to be recreated manually.

The component transformation process involves translating a scenario component of a given format to that of a different format. There are several problems that may arise during component transformation. Some of these may be insurmountable for the automated process and require human intervention.

The object model selected to represent scenario components is extremely simple. The component generation process has been design to allow the addition of new scenario source types without modification of existing source code. The component transformation process relies heavily on metadata to alleviate the need to represent transformation relationships in source code. These design decisions were made with flexibility of the overall design in mind.

Relevant component retrieval and component transformation are the two main functions of the semantic broker. Scenario source files that must be searched during the relevant component retrieval process may be distributed across multiple systems. The architecture of the semantic broker was designed with this in mind.

3.4 Semantic Broker Architecture

The architecture of the semantic broker is distributed to contend with the dispersed nature of the scenario source files. The semantic broker is divided into two applications, the *SemanticGateway* application and the *ScenarioRegistry* application. Software agents are utilized by both applications to request data, process signature-based queries, and receive replies. Figure 27 provides an application-level view of the semantic broker's two components. As the figure shows, there is one *SemanticGateway* application and multiple *ScenarioRegistry* applications. In fact, there is one *ScenarioRegistry* application for each system that contains source files to be searched during the relevant component retrieval process. In the following sections, the designs of these two applications are covered in detail.

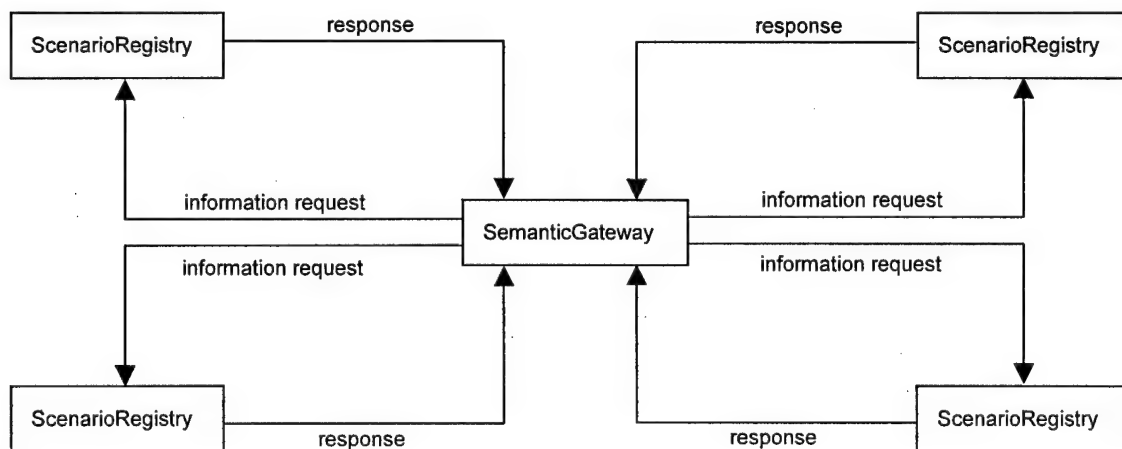


Figure 27: Application-Level View of Semantic Broker Architecture

3.4.1 ScenarioRegistry Application

In the semantic broker architecture, a *ScenarioRegistry* application executes on each system where scenario source files are registered. Figure 28 shows the major components and data sources of the *ScenarioRegistry* application. Excluded from this diagram are the *MetaComponent* and *SComponent* classes, which are used to create the component object trees. The functions of the *ScenarioRegistry* application are:

- Maintains a database of references to all registered scenario source files on the system. This database is its Source Registry Database (SRDB). The acronym SRDB and the term registry are used interchangeable throughout this work.
- Provides a Graphical User Interface (GUI) through which the user updates the contents of the application's SRDB.
- Accepts requests for its SRDB data, and returns the information to the requestor.
- Accepts signature-based queries, performs the relevant component retrieval, and returns its results to the requestor.

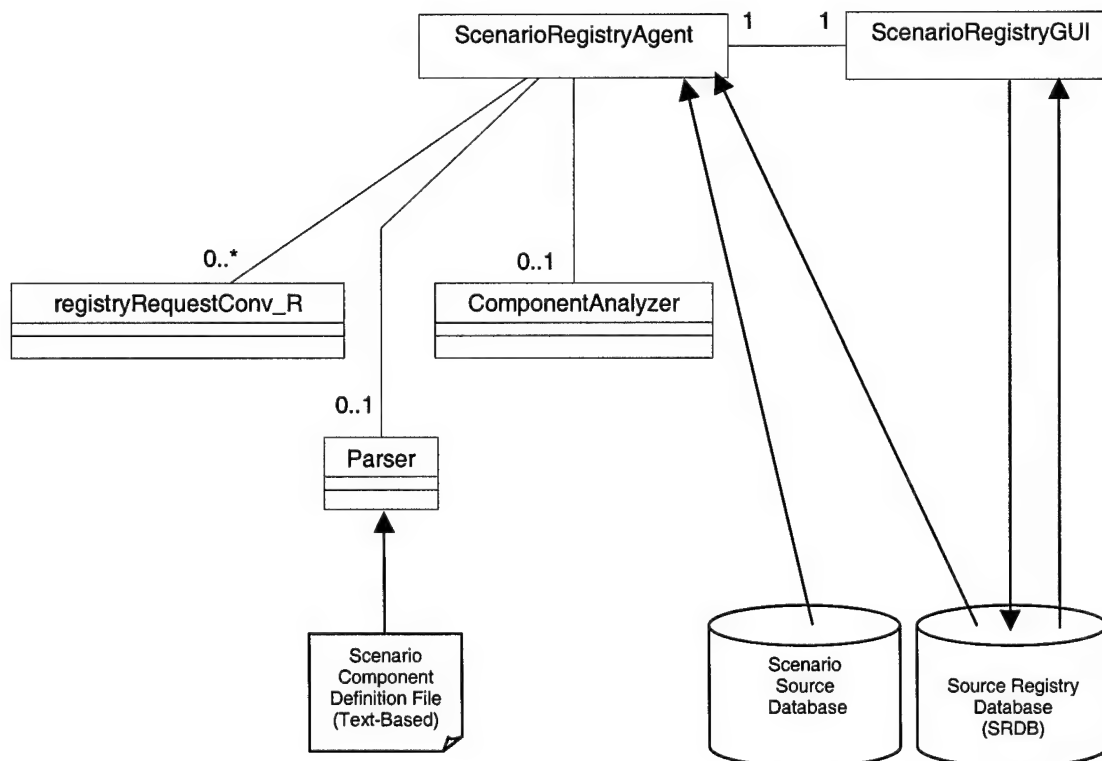


Figure 28: ScenarioRegistry Application Main Components and Data Sources

The design of the *ScenarioRegistry* application is divided in two main components:

- 1) The *ScenarioRegistryGUI*, which performs the first two functions listed; and 2) The *ScenarioRegistryAgent*, which performs the last two functions listed. These components will be discussed in turn.

3.4.1.1 *ScenarioRegistryGUI*

The *ScenarioRegistryGUI* portion of the *ScenarioRegistry* application provides a user interface to the contents of the application's SRDB. Through this interface, users register new scenarios and their files, and provide references to the metadata required to parse and analyze these files. The *ScenarioRegistryGUI* displays its registry's data in a manner consistent with the hierarchical nature of the data contained in the SRDB. The data in the SRDB allows the *ScenarioRegistry* application to track the origin of scenario source files. The origin information consists of abstract source (i.e., scenario) as well as physical source (i.e., path and filename).

The *ScenarioRegistryGUI* is the portion of the *ScenarioRegistry* application the user sees. When started, however, the user interface creates a *ScenarioRegistryAgent* and executes it on a separate thread.

3.4.1.2 *ScenarioRegistryAgent*

The *ScenarioRegistryAgent* is the workhorse of the *ScenarioRegistry* application. Objects of this type serve as the interfaces between each *ScenarioRegistry* application and the *SemanticGateway* application. *ScenarioRegistryAgent* objects accept requests for data, perform the necessary action to retrieve the data, and return the data to the requesting *SemanticGateway* application. One type of data request is a signature-based query. When a signature-based query is received, the component analysis process begins.

3.4.1.2.1 Component Analysis

When the *ScenarioRegistryAgent* receives a signature-based query, it creates a *registryRequestConv_R* object to respond to the requestor. The *registryRequestConv_R* object represents a conversation between agents. The *ScenarioRegistryAgent* then accesses the *Scenario Component Database* and retrieves the appropriate source component object models. For example, if the signature component's *type* is *SUPPRESSOR*, only component object models generated from registered *SUPPRESSOR* source files will be searched.

The *ScenarioRegistryAgent* utilizes the signature component to analyze scenario source files and determine their composition. Analysis results in the generation of a list of relevant scenario components. Each component is assigned a precision score—based on the number of sub-components and characteristics matched. Figure 29 shows how a signature component is utilized to analyze source components.

The analysis process begins with the *ScenarioRegistryAgent* creating a *ComponentAnalyzer* object and passing the selected signature component and the list of source components as parameters. *SComponent* objects know how to compare themselves to other *SComponent* objects and return a similarity score. The *ComponentAnalyzer* object calls the *analyzeComponent* method of each root present in its source component list and passes the signature component as the parameter. Each root's *analyzeComponent* method returns a similarity score based on its comparison of itself to the signature component. The *ComponentAnalyzer* object returns a list of those source components whose similarity score is greater than zero. The returned list is sorted on similarity score in descending order. The *registryRequestConv_R* object sends this sorted list to the requestor (i.e., the *SemanticGateway* application).

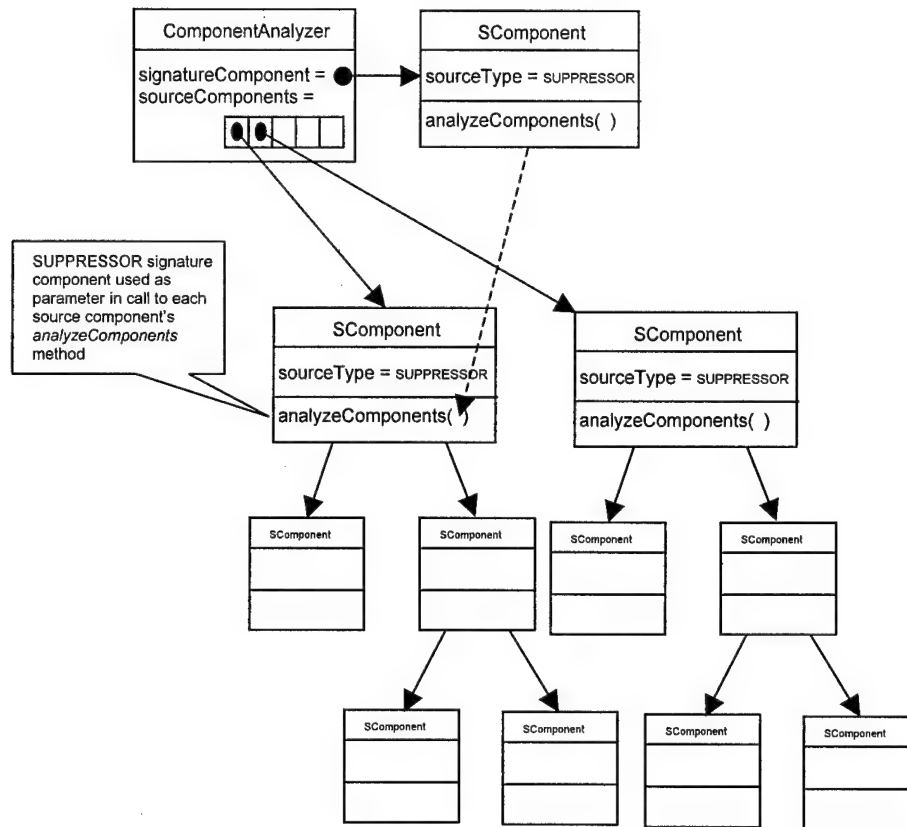


Figure 29: Component Analysis

3.4.1.2.2 Registry Forwarding

The other type of request for data the *ScenarioRegistryAgent* receives is a registry request. These requests are sent by the *SemanticGateway* application when it needs to update its system-wide source registry.

To respond to these requests, the *ScenarioRegistryAgent* creates a *registryRequestConv_R* object. This object calls the *getRegistry* method in its parent object (i.e., the *ScenarioRegistryAgent*), which returns the contents of the registry. The registry is then sent to the requestor.

The *ScenarioRegistry* application is responsible for registering and providing access to the scenario source files on a particular system. The functions of maintaining a

system-wide registry, signature selection, compiling relevant component search results, and component transformation fall into the realm of the *SemanticGateway* application.

3.4.2 SemanticGateway Application

The *SemanticGateway* application serves as the user's interface to the semantic broker. Through this application, a user selects a query signature and tailors its content, initiates the system-wide relevant component retrieval process, and transforms scenario components to a selected target format. Figure 30 shows the major components and data sources of the *SemanticGateway* application.

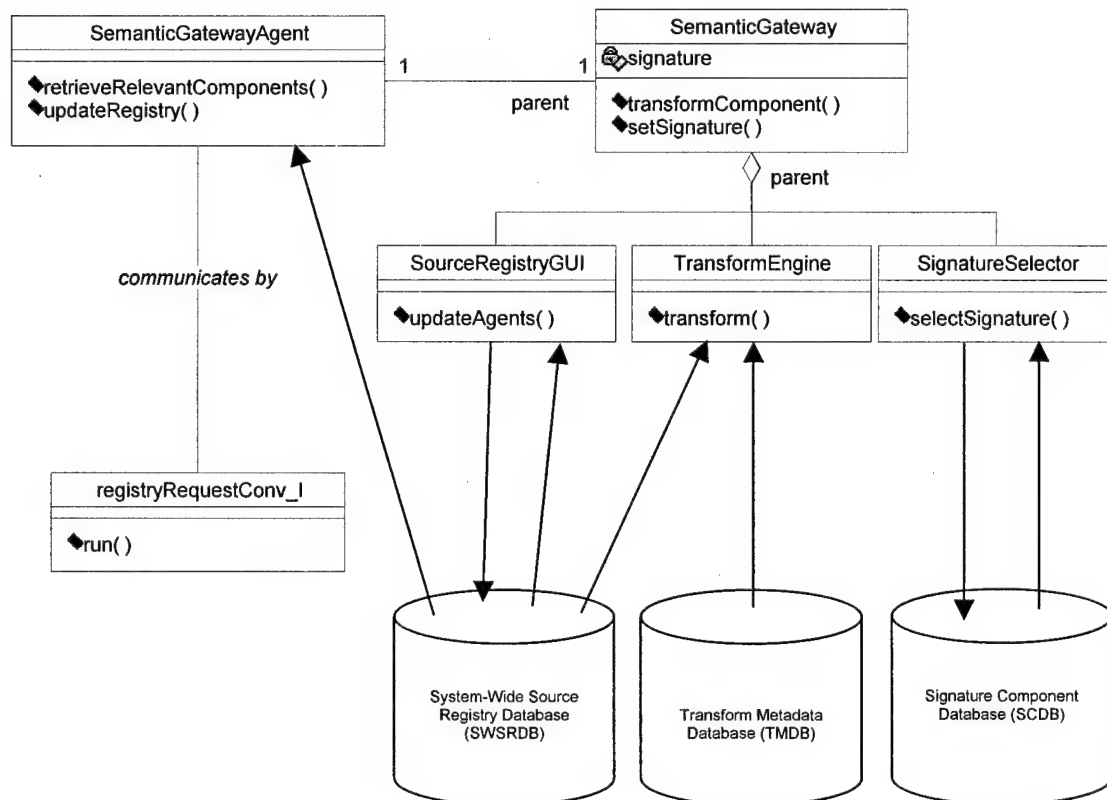


Figure 30: *SemanticGateway* Application's Major Components

Each *SemanticGateway* object has one *SemanticGatewayAgent* object, which it dispatches to update its *System-Wide Source Registry Database (SWSRDB)* and perform relevant component queries. Additionally, the *SemanticGateway* object creates

SourceRegistryGUI, *SignatureSelector*, and *TransformEngine* objects. These objects are used to provide a user interface to the source registry, enable editing and selection of signature components, and facilitate component transformation, respectively. These classes are discussed in greater detail in the following sections.

3.4.2.1 *SemanticGatewayAgent*

The *SemanticGatewayAgent* class provides the mechanism through which the *SemanticGateway* application interfaces with the multiple *ScenarioRegistry* applications distributed system-wide. The *SemanticGateway* object dispatches the *SemanticGatewayAgent* object to perform two operations critical to the semantic broker's overall functionality:

- 1) Requesting registry updates from each *ScenarioRegistry* application registered in the *SemanticGateway* application's *System-Wide Source Registry Database (SWSRDB)*.
- 2) Sending signature-based queries to each registered *ScenarioRegistry* application and compiling the relevant component result sets from each response.
- 3) Requesting relevant component details from *ScenarioRegistry* applications.

To perform any of these functions, the *SemanticGatewayAgent* creates one or more *registryRequestConv_I* objects. These objects initiate a conversation with each of the *ScenarioRegistry* applications referenced in the SWSRDB. This process is shown in Figure 31.

In the figure, the heavy dotted line between the *SemanticGatewayAgent* object and the *registryRequestConv_I* objects represents the fact that the *SemanticGatewayAgent* object created these objects and maintains a reference to them. There is one *registryRequestConv_I* object for each *ScenarioRegistry* application that must be contacted. Each of these objects sends a message to its assigned *ScenarioRegistry* application requesting the applicable service (i.e., registry update or relevant component

search). The *ScenarioRegistry* application performs the requested action and packages its result set in a reply message it sends to the *registryRequestConv_I* object that requested the service. When it has received a reply, the *registryRequestConv_I* object passes its result set to the *SemanticGatewayAgent* and terminates. After all conversations have terminated, the *SemanticGatewayAgent* consolidates the result sets and passes the aggregate set to the *SemanticGateway*.

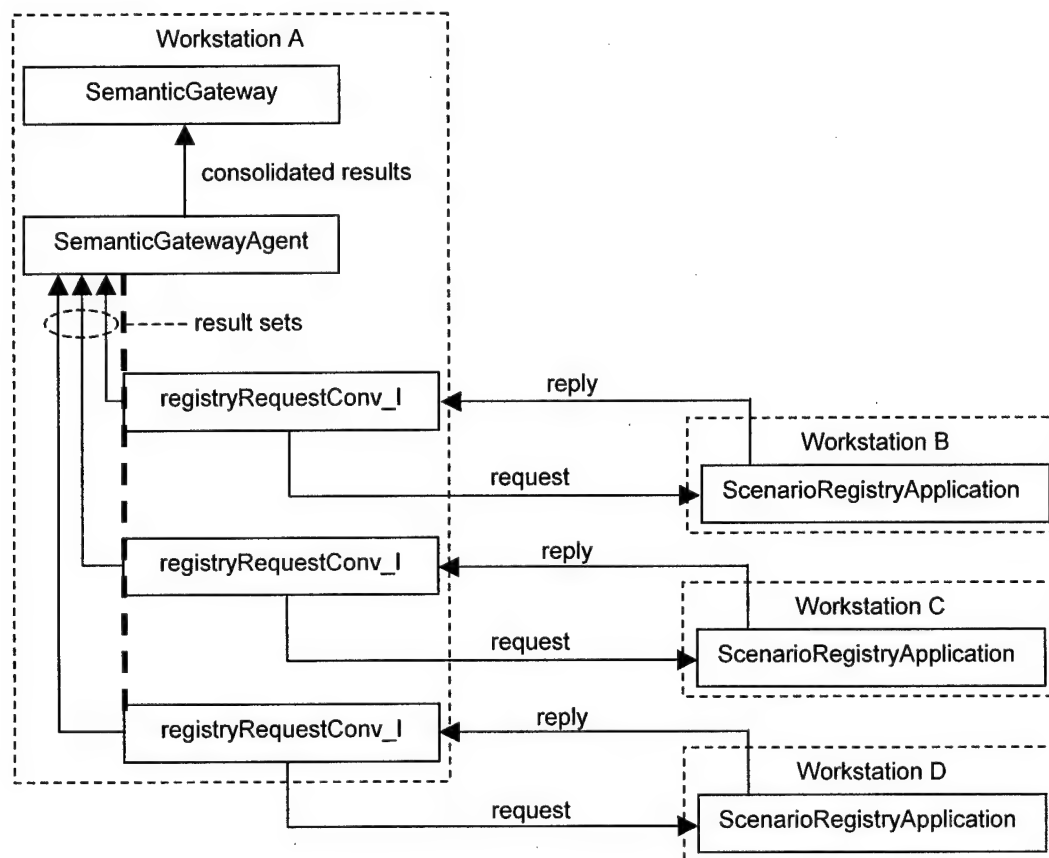


Figure 31: *SemanticGatewayAgent* Conversation Process

The *SemanticGatewayAgent* class works with the *ScenarioRegistryAgent* class to provide an interface between the *SemanticGateway* and *ScenarioRegistry* applications. In order to contact the *ScenarioRegistryAgent* objects, the *SemanticGatewayAgent* object relies on the scenario agent references stored in the SWSRDB.

3.4.2.2 System-Wide Source Registry Database (SWSRDB)

In order to maintain traceability of components, the semantic broker must be capable of tracking the source (i.e., the machine name, path, and filename) of each component generated. To facilitate this requirement, the *SemanticGateway* maintains the SWSRDB. This database contains the following information for each type of scenario source registered (e.g., SUPPRESSOR, SWEG, etc.):

- A reference to the file containing scenario component syntax definitions.
- A reference to the parsers used to generate signature and source components.
- A reference to transforms used in translating components from this source type to a specified target type.
- A reference to metadata used during component transformation.
- One reference (i.e., host and port number) to each *ScenarioRegistry* application that contains scenario source files of this type in its registry.

The information contained in the SWSRDB is updated by two different sources. First, each time the *SemanticGateway* application is started, it requests a registry update from each of the *ScenarioRegistry* applications registered in the SWSRDB. The responses from these requests are used to update the scenario information in the database. The failure of a registered application to respond, results in that application's registry entry being labeled as *UNAVAILABLE*.

The second means through which the SWSRDB receives updates is the *SourceRegistryGUI*. The *SourceRegistryGUI* class provides an interface through which users can update the contents of the SWSRDB. When a new simulator scenario type is added to the SWSRDB, the user must provide the metadata, parser, and transformation data necessary for the *SemanticGateway* to perform component generation, component analysis, and component transformations. Additionally, when a new *ScenarioRegistry* application is added to the system, its existence must be registered in the SWSRDB

before it will be recognized by the *SemanticGateway* application and utilized during the relevant component retrieval process.

The SWSRDB is a repository of the information required by the semantic broker to perform its functions. This database is updated automatically upon startup of the *SemanticGateway* application, and can be manually updated by the user at any time through the *SourceRegistryGUI*. The *SemanticGateway* application function that most heavily relies on the information in the SWSRDB is component transformation.

3.4.2.3 Component Transformation

The translation of a scenario component from its source format to a target format of choice is known throughout this work as component transformation. In the *SemanticGateway* application, component transformation is initiated by the user through the *SemanticGateway* object. This object references the SWSRDB to obtain references to transform metadata files and transform classes, and passes this data to a newly instantiated *TransformEngine* object. The *TransformEngine* class is the workhorse of component transformation in the *SemanticGateway* application. Figure 32 shows the classes and data sources involved in the component transformation process, and a description of the process follows.

To facilitate all the different categories of transformations presented in Section 3.3.4, Figure 26, the *SemanticGateway* application creates a *TransformEngine* object, and passes it the transform metadata reference mentioned above. The *TransformEngine* object creates a *TransformMDParser* object and parses the metadata file containing the transform metadata for the source-to-target transformation. This metadata file is contained in the *Transform Metadata Database* (TMDB). During the initialization process, the *TransformEngine* also instantiates the *Transform* objects referenced in the

metadata and maintains a reference to each of these. Next, the *TransformEngine* object processes each *SComponent* object beginning with the root. Based on transform category information supplied through the transformation metadata, the *TransformEngine* object accesses and utilizes the appropriate sub-class of *Transform* object. Collectively, the *Transform* objects handle the translation of the component from source to destination format, and return the resulting *SComponent* object to the *TransformEngine* object.

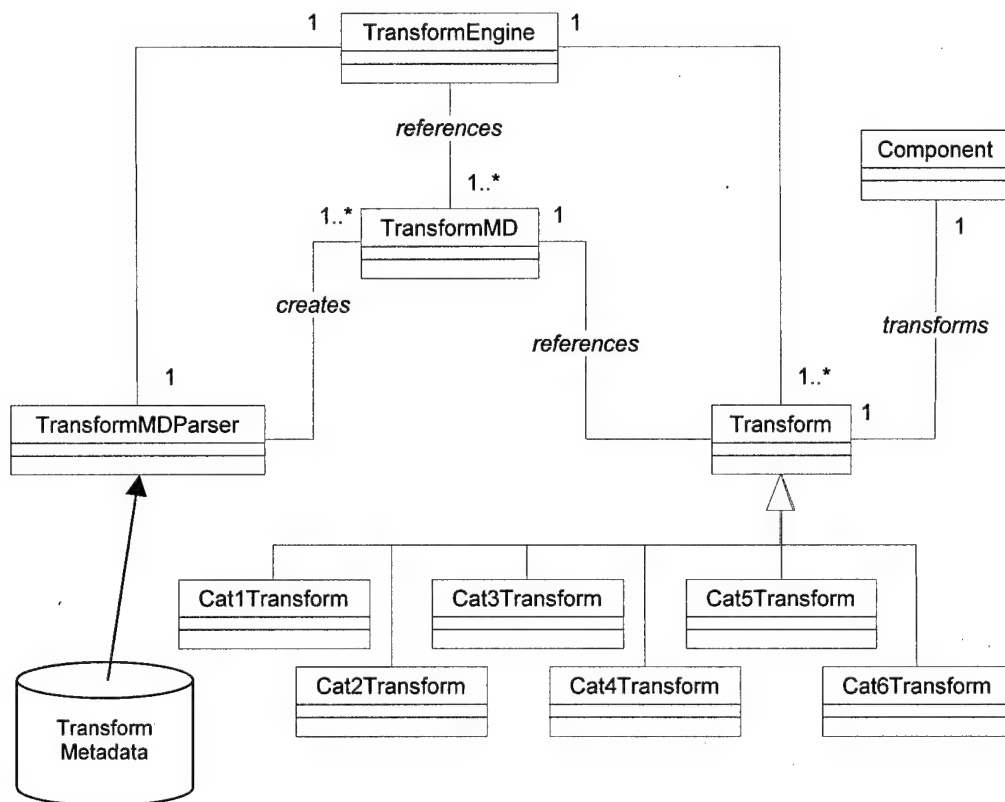


Figure 32: Component Transformation Classes and Data Source

Extensibility in the component transformation portion of the semantic broker is achieved by providing a set of *Transform* sub-classes that extend the *Transform* class and define the process of transformation for each of the cases identified in Section 3.3.4, Figure 26. Additionally, the inclusion of a new simulator scenario source requires that metadata be provided in the TMDB.

The TMDB shown in Figure 32 maps a component type to one of the categories listed in Figure 26. The data in the TMDB effectively maps each component type to a specific transform category, and provides the necessary details to allow the *Transform* object to perform the translation. Figure 33 provides the transform metadata file format for the first three transformation categories. In Figure 33, an asterisk represents the fact that there may be one or more of an item or group, and square brackets are used to group two or more items. The last three transformation categories represent the most difficult aspects of the transformation process, and are not detailed here. The fourth category requires some *Transform* object implementation specific logic in order to perform the transformation. The fifth category is the most difficult to implement, since the software must determine whether user intervention is required and, if so, must present the user with the problem and recommend possible solutions. This category is beyond the scope of this work, and such transformation will be handled in the same way as Category 6 transformation. Category 6 transformations, as implemented in this work, provide comments in the target scenario stating which component of the source scenario could not be translated.

```
CATEGORY 1
<component type>*
END CATEGORY 1
CATEGORY 2
[ <component type>
<keyword not in source>*
END <component type> ]*
END CATEGORY 2
CATEGORY 3
<component type>
[ <source keyword> <destination keyword> ]*
END <component type>
END CATEGORY 3
```

Figure 33: Transform Metadata File Format

3.5 Summary

This chapter begins with a review of the tools used in the analysis, design, and implementation phases of this research effort. Next, a discussion of the general approach used to design the Semantic Broker in this research covers the topics of scenario component representation, relevant component retrieval, and component transformation. Finally, the design of the semantic broker is presented in some detail including the classes necessary for its implementation and the data sources it will utilize.

This chapter outlines the semantic broker as it will be implemented in this research. In Chapter 4, the *SemanticGateway* and *ScenarioRegistry* applications are developed and tested.

4. IMPLEMENTATION

4.1 Introduction

This chapter discusses the functionality of the Semantic Broker as implemented in this research. The resulting tool is presented primarily as a proof of concept vehicle with minimal intent to maximize the efficiency of the tool's algorithms or data structures. The chapter begins with a discussion of some design issues that were encountered during detailed design and implementation of the Semantic Broker. Next, the two main components of the broker, the *SemanticGateway* and *ScenarioRegistryGUI* applications, are discussed. These two components provide all the functionality necessary to provide scenario source registration, relevant component retrieval, and scenario component transformation. This is followed by a review of test data collected concerning the component retrieval portion of the tool. Finally, the requirements for extending the Semantic Broker to include additional simulation types (e.g., JIMM, EADSIM, etc.) are discussed.

4.2 Design Issues

During implementation, three key issues came to light that are worthy of mention here. These include the class hierarchy used to represent scenarios and its reliance on metadata, scenario component generation, and signature analysis.

4.2.1 Scenario Class Hierarchy

The Semantic Broker's design utilizes a simple class hierarchy to represent scenario components and relies on metadata to interpret scenario source files and transform selected components. The class hierarchy utilized in this research to represent scenario

components is shown in Figure 34. This simple class hierarchy allows one object model to be used to build object representations of any simulator's scenario components.

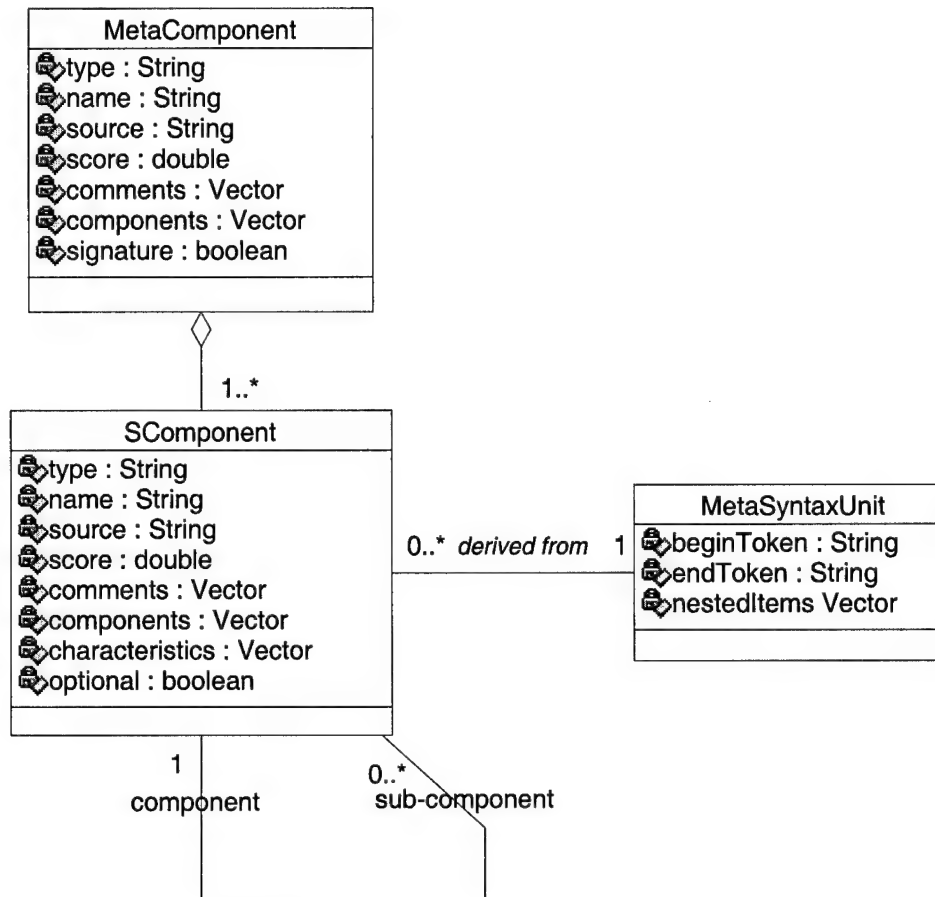


Figure 34: Semantic Broker Scenario Component Class Hierarchy

This model is used to represent both signature and source components. Signature components are essential for signature analysis, which is covered in Section 4.2.3. However, there is a difference in the structure of the object trees of signature components vs. source components. The root of a signature component object tree is a *MetaComponent* object. Every other node in the signature object tree is an *SComponent* object. The role of the *MetaComponent* class in relation to signature components is that of a descriptor. The *MetaComponent* object root of each signature component contains a *name* attribute that provides a descriptive name for the signature,

and a *comments* attribute that contains comments that further describe the signature component, its capabilities, and its limitations.

A source component object tree is composed entirely of *SComponent* objects. However, the *MetaComponent* class has a role in relation to source component trees also. Here the *MetaComponent* object represents an entire collection (source file) of source components. The *MetaComponent* object's *components* attribute contains a vector of pointers to the root *SComponent* object for each source component generated by the file parser. Essentially, the role of the *MetaComponent* class in relation to source components is to represent a scenario source file and provide links to the scenario source components defined within it.

The attributes of the *MetaComponent* and *SComponent* classes have similar names, and one may consider simply making the *SComponent* class extend the *MetaComponent* class. However, doing so would violate sound software engineering principles, since these two classes lack the *IS A* relationship. Because of this, although the attributes have similar names, some have different meanings. The attributes of the *MetaComponent* class are listed below. Attributes that have the same name and meaning as an attribute of the *SComponent* class are annotated as such and are not repeated in the paragraph describing the *SComponent* class' attributes.

- **type:** The type of scenario source or signature component (i.e. SUPPRESSOR, SWEG, etc.).
- **name:** This attribute contains the name of the file from which the scenario or signature component was generated.
- **score:** This attribute is the score assigned to the object during signature analysis. *SComponent* attribute *score* has the same meaning. See Section 4.2.3 for details on signature analysis.
- **source:** This is the location of the scenario source file from which the object was generated. For signature components, this attribute is null. *SComponent* attribute *source* has the same meaning.

- **comments:** This attribute contains comments about the capabilities and limitations of the component. *SComponent* attribute *comments* has the same meaning.
- **components:** This attribute contains a set of sub-components for this object. *SComponent* attribute *components* has the same meaning.
- **signature:** This attribute is set to *true* if the object is a signature component.

As mentioned previously, most of the attributes of the *SComponent* class have the same names as the attributes of the *MetaComponent* class. However, the meaning of two of the *SComponent* class' attributes differ significantly as outlined below.

- **type:** The type of scenario component represented (PLAYER-STRUCTURE, COMM-RCVR, etc.).
- **name:** The name of the scenario component as defined in the scenario source file.
- **optional:** This attribute is checked during signature analysis to determine if a signature component or sub-component is mandatory. The value of this attribute is true by default and is irrelevant in source component objects.

The simplicity of this object model facilitates the modeling of virtually any simulation scenario source type. However, this approach requires extensive use of metadata to determine how to interpret source files during parsing. This metadata is provided through instances of the *MetaSyntaxUnit* class.

MetaSyntaxUnit objects are utilized by file parsers to determine the correct interpretation of scenario components as they are encountered in the scenario source file. There is a *MetaSyntaxUnit* object for each type of scenario component and sub-component that can possibly be represented in the source scenario format. Each *MetaSyntaxUnit* object contains the following attributes:

- **beginToken:** The parser uses this attribute to determine that a new scenario component, or sub-component, definition follows. The value of this attribute is also the value of the *type* attribute of the new *SComponent* object that will be created to represent this scenario component.
- **endToken:** The parser uses this attribute to determine when the end of the scenario component definition has been reached. In some cases, this attribute

will have the value *null*, which indicates that the scenario component's definition has no terminating token. In these cases, the parser uses the fact that it has encountered a valid *startToken* to determine it has reached the end of the current scenario component's definition.

- ***nestedItems***: This attribute contains the types of items that may be nested within the begin and end tokens of the specified scenario component definition.

The scenario source file parser maintains a list of these *MetaSyntaxUnit* objects. It builds this list from a text-based *meta syntax* file that contains syntax definitions for each type of scenario component that can exist in a scenario source file. There is a *meta syntax* file for each scenario source type (e.g., SUPPRESSOR, SWEG, EADSIM, etc.). A portion of the syntax file for SUPPRESSOR scenario source files is shown in Section 3.5.2.2, Figure 30. *MetaSyntaxUnit* objects are the key to the parser's ability to perform component generation.

4.2.2 Component Generation

SComponent objects are instantiated by a component generator designed specifically to parse scenario source files of the specified type and create object trees that represent the scenario component definitions in the file parsed. The component generator references *MetaSyntaxUnit* objects to determine how to interpret components as they are read from the file. In the Semantic Broker system, all component generators must extend the abstract class *Parser*.

The *Parser* class provides a method, *loadMetaSyntax*, to parse the file containing the syntax structure for the scenario source file. The syntax structure is provided in a text-based file. From this file, *MetaSyntaxUnit* objects are instantiated. The component generator class must implement the *Parser* class's abstract method *generateComponents*. This method references the data structure containing the *MetaSyntaxUnit* objects to determine how to instantiate each scenario component contained in the source file being parsed. The *generateComponents* method returns a

Java Vector containing the root objects of all scenario components contained in the target source file. Component generators instantiate object model representations of scenario components for both scenario sources and new signature components being added to the *Signature Component Database* (SCDB).

4.2.3 Signature Analysis and the SCDB

The Semantic Broker, as designed and implemented in this research, utilizes signature analysis to identify scenario components that may be of interest to the user. The signatures are object representations of generic components of a specific scenario type. Essentially, a signature component is a complex query structure. A signature can be modified by the user to represent scenario components of varying levels of detail. For example, the user might select a signature component named "Bomber" from the list of available signatures, then modify the signature's sub-components and characteristics to match the search criteria. The user could then use the Semantic Broker to retrieve source components that match the search criteria (i.e., the signature).

Since signatures are object representations of generic components of a specific scenario type, there is a separate set of signatures for each type of scenario source (e.g., SUPPRESSOR, SWEG, etc.) for which the system is capable of searching. The SCDB contains a set of object model representations of signature components for the various simulator scenario types the system is capable of analyzing.

The function of signature analysis is performed by the *ComponentAnalyzer* class in conjunction with the *analyzeComponents* method of the *MetaComponent* class and the *analyzeComponents* and *analyzeAttributes* methods of the *SComponent* class. These classes and their inter-relationships are shown in Figure 35.

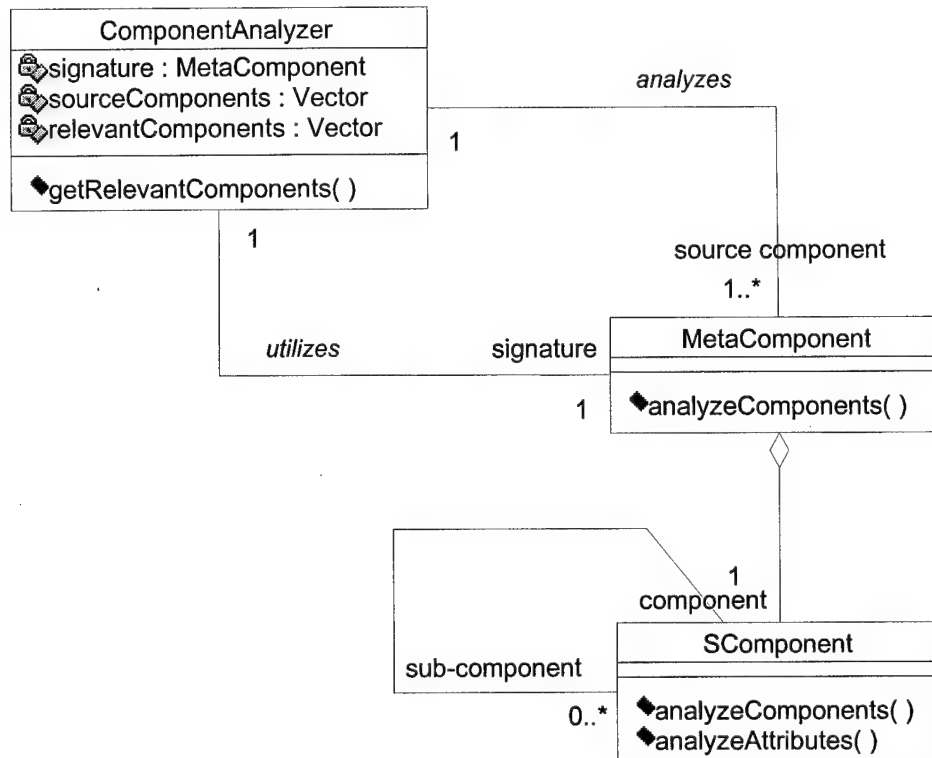


Figure 35: Signature Analysis Classes

The constructor for the *ComponentAnalyzer* requires two parameters: a *MetaComponent* signature object, a Java Vector containing a *MetaComponent* object for each set of source components to be analyzed. After instantiating the *ComponentAnalyzer* object, its *getRelevantComponents* method is called to initiate the relevant component retrieval process. This method iterates through the vector of *MetaComponent* objects and calls each object's *analyzeComponents* method with the signature's root *MetaComponent* object as the parameter. This method, in turn, calls the *analyzeComponents* method of each of the *SComponent* objects in its *components* vector with the root *SComponent* object of the signature as the parameter. At this point, the process becomes recursive: the *SComponent* object's *analyzeComponents* method

first calls the object's *analyzeAttributes* method. This method returns a score based on the number of characteristics that match the signature *SComponent* object's characteristics. The *analyzeComponents* method then iterates through the input signature *SComponent* object's components attribute. For each of the signature's sub-components, this method searches the source component's vector of sub-components to determine if the source component has a matching type of sub-component. If so, this sub-component's *analyzeComponents* method is called with the signature sub-component as the parameter. This recursive process continues until the leaf nodes of the signature component's object model have been analyzed.

The *SComponent* object's *analyzeComponents* method returns a *double*. The highest score possible is one and indicates that the source component was the same type (e.g. PLAYER-STRUCTURE, COMM-RCVR, etc.) as the signature, and has a sub-component object tree that includes all the sub-components of the signature. A score of one does not, however, mean the source component is an exact match to the signature component. It indicates that the signature component's object model is a subset of the source component's model. This requirement can be formally stated as $\forall \alpha (\alpha \in S \wedge \alpha \in R \wedge \lambda \subseteq \alpha)$, where λ is the signature, S is the applicable set of scenario source files, and R is the set of relevant components. The set of relevant components is compiled by each of the *MetaComponent* objects and returned to the *ComponentAnalyzer* object, which aggregates all relevant components and returns the entire set.

The implementation of the Semantic Broker uses a simple object model to represent scenario components. This simplicity provides the flexibility necessary to allow the model to represent virtually any scenario source type. Specialized parsers are utilized to create object representations of scenario components from text definitions—a process known as *component generation*. Signature analysis is utilized to facilitate the retrieval

of relevant scenario components from the available source files. Signature components are stored in the Signature Component Database. These functions are performed by different components of the Semantic Broker.

4.3 Semantic Broker Major Components

The design of the Semantic Broker utilizes multi-agent technologies to provide signature-based search capability across distributed platforms. The use of the Java programming language enhances the ability of the system to operate in a heterogeneous environment. The Semantic Broker system is divided into two main components: the *SemanticGateway* application, and the *ScenarioRegistry* application. Figure 36 provides a system-level view of how these two components interact.

Through the *SemanticGateway* application's GUI, the user builds a signature-based query to specify, as generically as possible, the kinds of scenario components that are of interest. As shown in the figure, the *SemanticGateway* application sends this search criteria to each *ScenarioRegistry* application. The *ScenarioRegistry* applications provide an interface to all scenario source files present on systems B, C, and D. This interface is dual faceted. First, the *ScenarioRegistry* application provides a GUI through which users register scenarios and their associated source files. Second, the *ScenarioRegistry* application provides an agent-based interface to all scenarios contains in its registry. When a *ScenarioRegistry* application receives a relevant component query, it searches its applicable scenario source object models and responds with a set of references to the relevant components it contains in its *Scenario Component Database*.

Once the references to the relevant components have been received, the *SemanticGateway* application allows the user to view each reference's comments and retrieve details on a selected component. Requesting the details of a component

reference causes the *SemanticGateway* application to send a request to the appropriate *ScenarioRegistry* application requesting the details of the component. The *ScenarioRegistry* application responds by returning the entire object model of the component to the *SemanticGateway* application. After the details have been received, the user may expand each level of detail to determine if the component is suitable for inclusion in a new scenario. If the selected component is a different type (i.e., SUPPRESSOR vs. SWEG) than the scenario under construction, the user may, with certain limitations, transform the component to the target format.

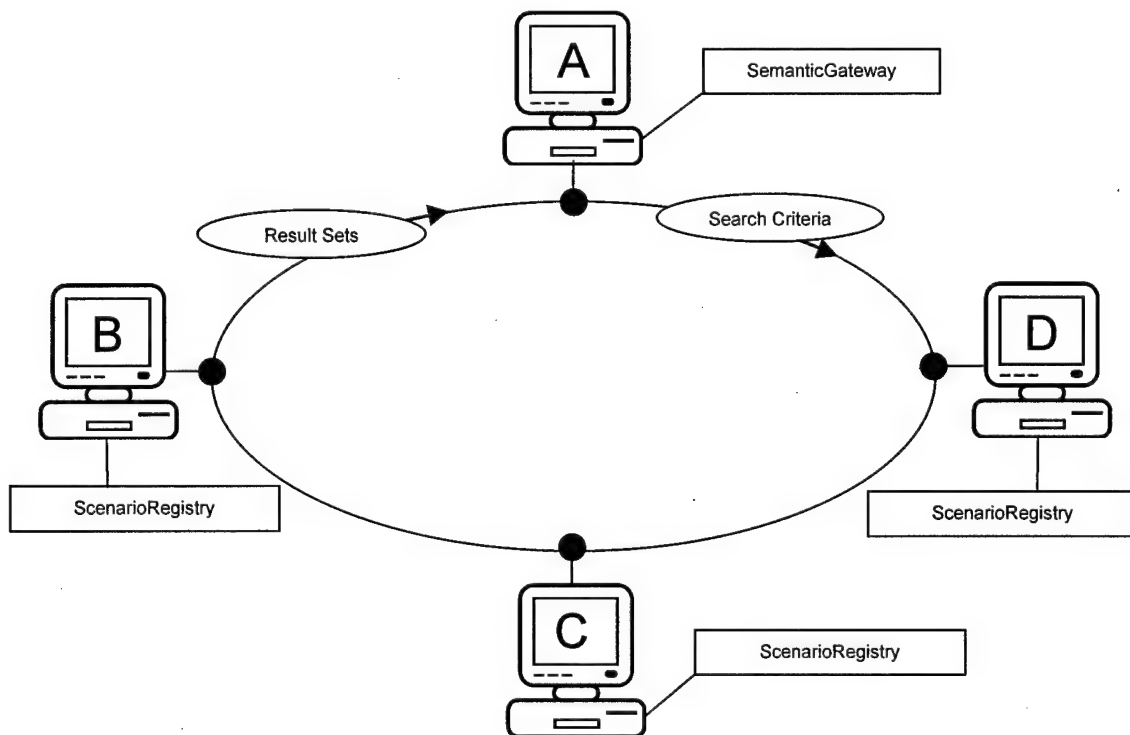


Figure 36: Semantic Broker System-Level View

Both the *SemanticGateway* and *ScenarioRegistry* applications utilize agent technology to perform various functions. The following sections detail the implementation of these applications.

4.3.1 ScenarioRegistry Application

The *ScenarioRegistry* application provides an interface, for both the user and the *SemanticGateway* application, to all scenario source files referenced in its registry. Figure 37 provides a class diagram for the *ScenarioRegistry* application. The components of the *ScenarioRegistry* application perform the following Semantic Broker functions:

- Provide their registry contents on request from the *SemanticGatewayAgent* object. This allows the *SemanticGateway* application to update its system-wide registry of available scenario registry agents.
- Receive signature-based queries from a *SemanticGatewayAgent* object, perform the query on their registered scenario source files and respond with the set of references to the relevant components generated by the query.
- Respond to requests for the details of a relevant component reference by replying with a message containing the entire object model representation of the relevant component.
- Provide a GUI that allows the user to register scenarios and their source files.

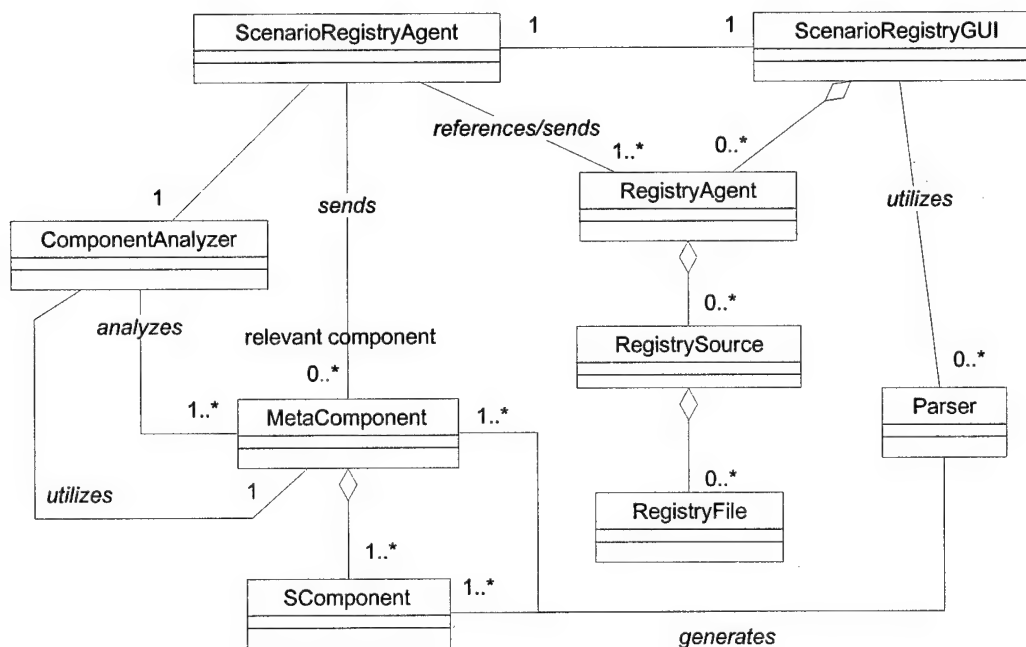


Figure 37: ScenarioRegistry Application Class Diagram

As the figure shows, there is one *ScenarioRegistryAgent* object for each *ScenarioRegistry* object. The *ScenarioRegistryAgent* class provides a machine-to-machine interface, via network communications, for the *ScenarioRegistry* application. The *ScenarioRegistryGUI* class provides a human-to-machine interface to the scenario registry. These two components are covered in detail in the following sections.

4.3.1.1 *ScenarioRegistryAgent* Class

The *ScenarioRegistryAgent* class extends the *Agent* class of the *AFIT Agent MOM* agent architecture. There is one *ScenarioRegistryAgent* object for each parent *ScenarioRegistryGUI* object. The *ScenarioRegistryAgent* object executes in a separate thread from its parent and calls methods in its parent to retrieve information in response to registry requests and relevant component queries. The *ScenarioRegistryAgent* essentially acts as an information server. It monitors the port specified by the user when the *ScenarioRegistry* application was started. When it receives a request, it spawns conversation objects on separate threads and these conversation objects respond to the request. Figure 38 shows the class hierarchy of the *ScenarioRegistryAgent* and *registryRequestConv_R* classes.

As the figure shows, these “requests” come in the form of *Message* objects. The *Message* class is part of the *AFIT Agent MOM* architecture as are the *Agent* and *Conversation* classes. The *ScenarioRegistryAgent* and *registryRequestConv_R* classes extend the *Agent* class and *Conversation* class respectively. When a *Message* object is received, the *ScenarioRegistryAgent* creates a *registryRequestConv_R* object to respond. The constructor of this object is passed the *Message* object, a parent object (the agent that created it), an *ObjectInputStream* object, and an *ObjectOutputStream* object. Then the *run* method of the object is called. Based on the value of the

performative attribute of the *Message* object, the *registryRequestConv_R* object calls the appropriate method in its parent object to obtain the result set it requires. Once the result set has been received, this object writes a *Message* object containing the result set to the *ObjectOutputStream*.

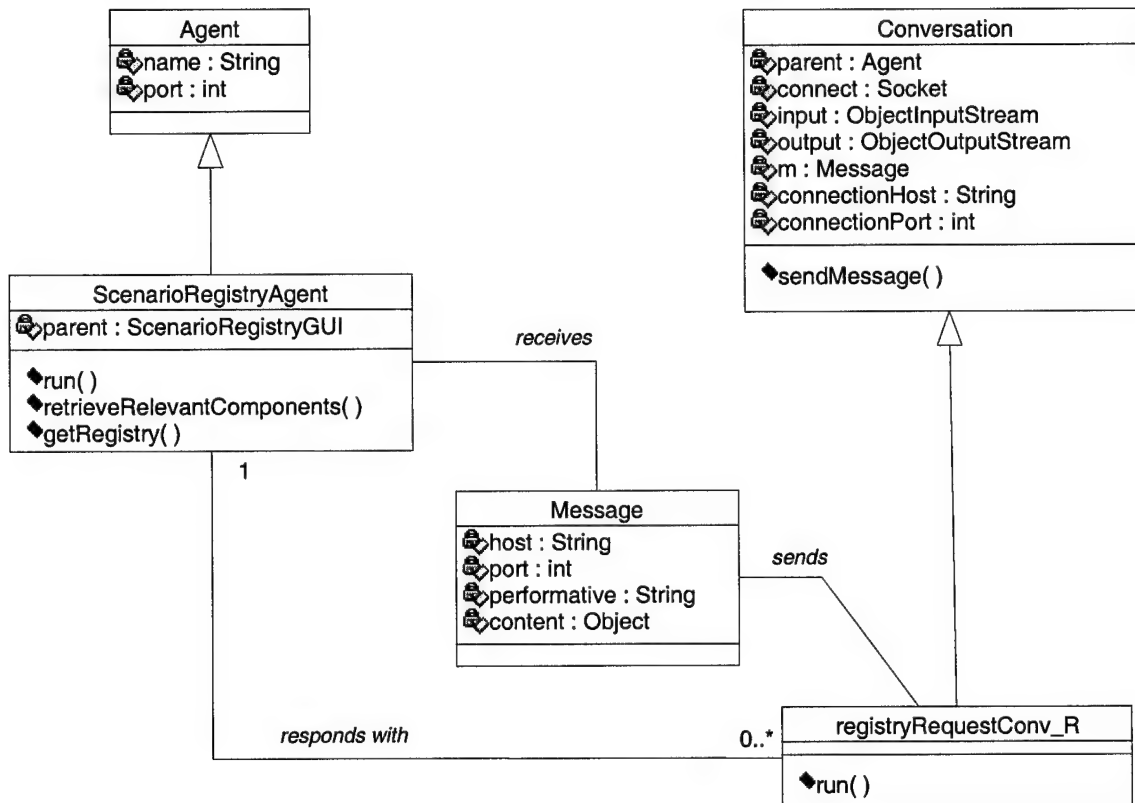


Figure 38: *ScenarioRegistryAgent* Class Hierarchy

The *ScenarioRegistryAgent* acts as an interface to the scenario registry through which the *SemanticGateway* application can request registry information, signature-based queries, and relevant component details. Essentially, the *ScenarioRegistryAgent* provides the machine-to-machine interface to the object model representation of the scenario source files on the *ScenarioRegistry* application's host machine. The human-to-machine interface to these resources is provided by objects of the *ScenarioRegistryGUI* class.

4.3.1.2 ScenarioRegistryGUI Class

The *ScenarioRegistryGUI* provides the user interface to the scenario source files in the scenario registry. Objects of this class permit a user to register scenario source type agent entries and add scenarios and their associated files. Figure 39 shows the class diagram for the data contained in the scenario registry.

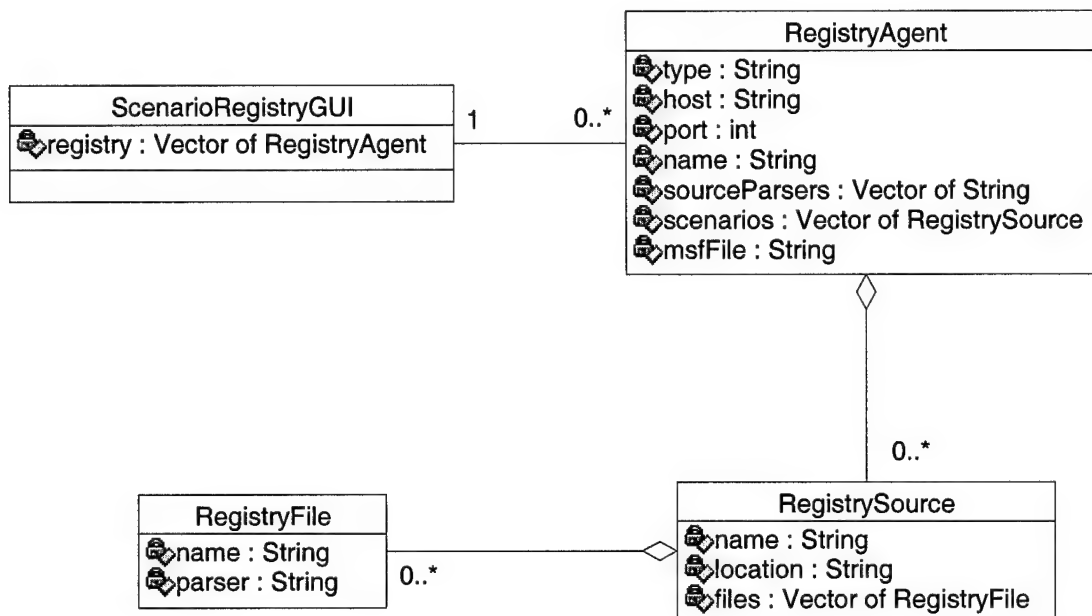


Figure 39: Scenario Registry Data Class Diagram

The *ScenarioRegistryGUI* class' attribute *registry* is a Java Vector that contains instances of the *RegistryAgent* class. There is a *RegistryAgent* object for each scenario source type registered in the registry database. The attributes of the *RegistryAgent* class are:

- **type**: This attribute contains the type of scenario source files (e.g. SUPPRESSOR, SWEG, etc.) referenced by the *RegistryAgent* object.
- **host**: The name of the system on which this *ScenarioRegistryAgent* operates.
- **port**: The port number on which the *ScenarioRegistryAgent* operates.
- **name**: The name of the *RegistryAgent* object.

- **sourceParsers:** This attribute is a *Vector* containing *String* objects that reference scenario source file parser classes. Java's reflection tools are utilized to instantiate objects from these strings at runtime.
- **msfFile:** This attribute is a *String* object that contains the name of the meta syntax file containing the scenario component syntax definitions for the type of scenario source files this *RegistryAgent* object references.
- **scenarios:** This attribute is a *Vector* that contains *RegistrySource* objects.

A *RegistrySource* object represents a simulation scenario, and there is one *RegistrySource* object for each scenario registered with a *RegistryAgent* object. The attributes of the *RegistrySource* class are:

- **name:** This attribute is a *String* object that contains the name of the scenario.
- **location:** This attribute is a *String* object that contains the path to this scenario's files.
- **files:** This attribute is a *Vector* that contains *RegistryFile* objects.

A *RegistryFile* object represents a source file for a given scenario. *RegistryFile* objects have two attributes:

- **name:** This attribute is a *String* object that contains the filename of the referenced scenario source file.
- **parser:** This attribute contains the class name of the *Parser* that knows how to generate components from the scenario source file referenced by this *RegistryFile* object. Java's reflection tools are used to instantiate the appropriate *Parser* object from the string contained in this attribute.

The *ScenarioRegistry* application is initialized by executing the *main* method of the *ScenarioRegistryGUI* class and providing the desired port number as a command-line argument. The *main* method creates a *ScenarioRegistryGUI* object. The *ScenarioRegistryGUI* class extends the *java.swing.JFrame* class and its constructor performs the following initialization functions:

- Creates a *ScenarioRegistryAgent* object and executes it on a separate thread.
- Loads its stored registry contents into memory.
- Generates a *JTree* representation of the registry's contents.

The *main* method then displays the *ScenarioRegistryGUI* object by calling its *show* method. This causes the window shown in Figure 40 to be displayed.

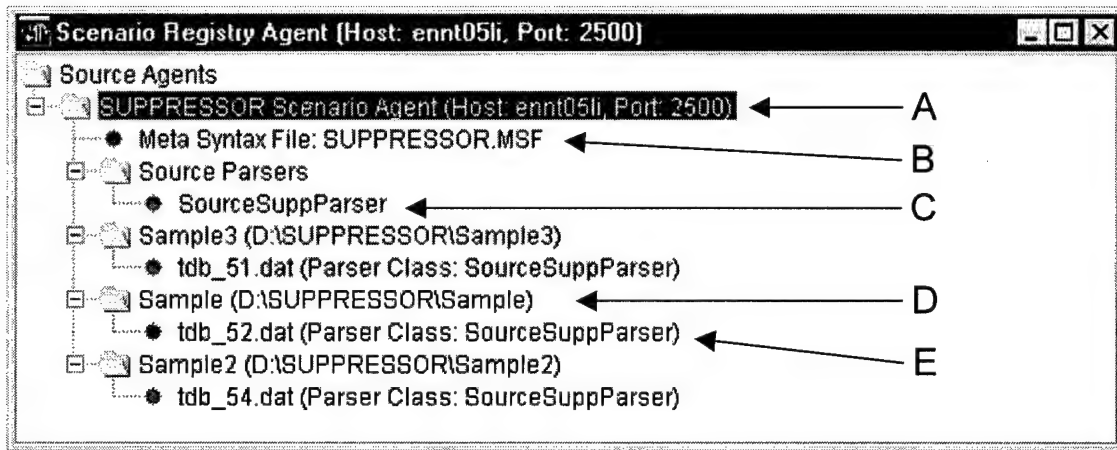


Figure 40: *ScenarioRegistryGUI* Window

As shown in the figure, the *ScenarioRegistryGUI* displays the contents of its registry in a *JTree*. The title area of the window displays the name of the host machine and the port the *ScenarioRegistryAgent* object is monitoring. In the figure, the nodes of the *JTree* have been labeled to allow a cross-reference between this figure and the objects of the underlying data structure shown in Figure 39.

- A. This node was generated from a *RegistryAgent* object contained in the *ScenarioRegistryGUI* object's registry Vector. The fact that there is only one child node at this level indicates that there is only one *RegistryAgent* object in the registry Vector.
- B. This node displays the value of the *msfFile* attribute of the *RegistryAgent* object referenced by A.
- C. This node references a source parser class name. This node was generated from the *String* object(s) contained in the *RegistryAgent* object's *sourceParsers* Vector.
- D. This node displays the contents of a *RegistrySource* object.
- E. This node displays the contents of a *RegistryFile* object. The class name of the *Parser* object that will be used to parse the file referenced is displayed in parentheses after the filename.

The user can update the contents of the registry by selecting an item with the mouse and clicking the right mouse button to display a popup menu. Through this menu, the user can add, delete and modify *RegistryAgent*, *RegistrySource*, and *RegistryFile* objects.

The *ScenarioRegistry* application provides an interface to the scenarios referenced in its registry. It responds to requests for its registry data and signature-based queries, and sends the specified data to the requestor. This is a portion of the overall functionality of the Semantic Broker. The functions of signature selection, system-wide scenario source registration, aggregation of all relevant component queries, and component transformation lie in the *SemanticGateway* application.

4.3.2 *SemanticGateway* Application

The *SemanticGateway* application is the core component of the Semantic Broker, and provides access to the system-wide source registry, signature selector, component retrieval, and transformation of selected relevant components. Figure 41 shows a high-level class diagram for the *SemanticGateway* application. The *SemanticGatewayAgent* *SourceRegistryGUI* classes are covered in detail in subsequent sections, and the *TransformEngine* and *ComponentViewer* classes are discussed in Section 4.3.5, which covers component transformation. The *SourceRegistry* class is covered in Section 4.3.2.2 in conjunction with the *SourceRegistryGUI* class.

Notably absent from this diagram is the *ComponentAnalyzer* class. *ComponentAnalyzer* objects are utilized to perform a signature-based query on a collection of scenario components and determine which, if any, should be included in the set of relevant components returned to the user. Relevant component retrieval is a core function of the semantic broker; however, this process is conducted in each of the

ScenarioRegistry applications referenced by the *RegistryAgent* objects in the system-wide source registry.

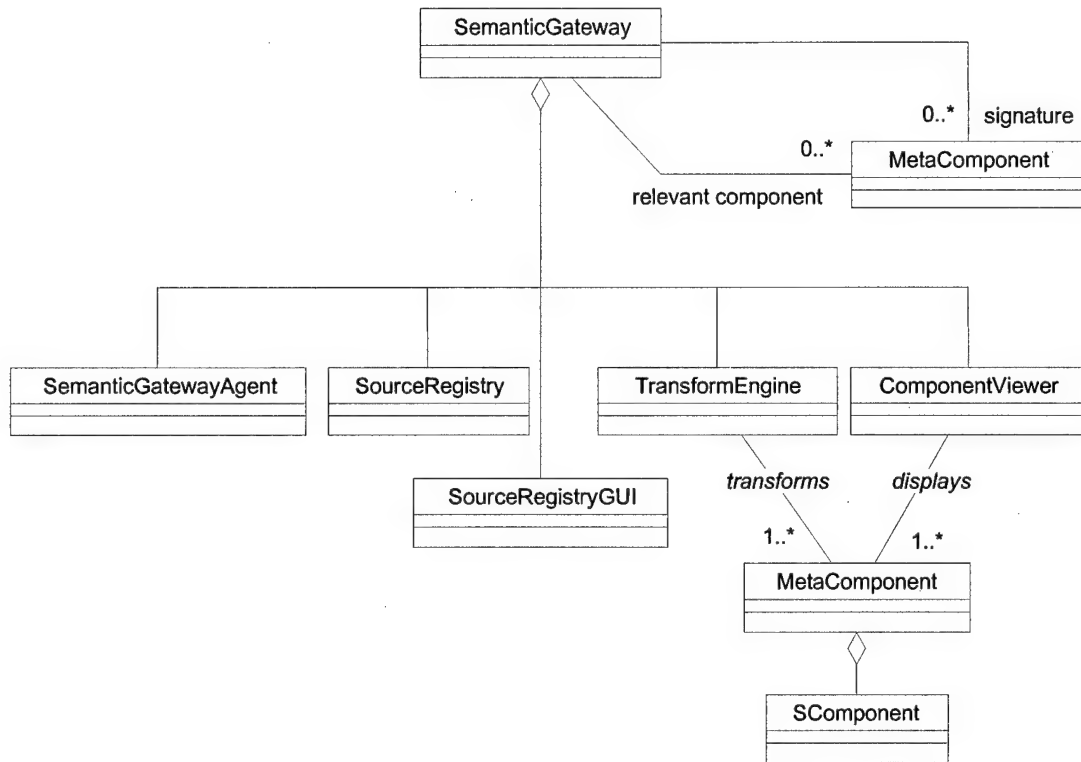


Figure 41: *SemanticGateway* Application Class Diagram

Figure 42 shows the *SemanticGateway* application window at initialization. The title area of the window displays the name of the host machine and the port the *SemanticGatewayAgent* is monitoring. In the figure, the upper half of the window is divided into two panes. The upper pane displays the list of relevant components returned in response to a signature component based query. The lower pane displays the comments associated with the selected component or sub-component. The lower half of the window displays the currently selected signature. The menu bar at the top of the window provides access to the source registry, signature selection, component retrieval, and component transformation functions.

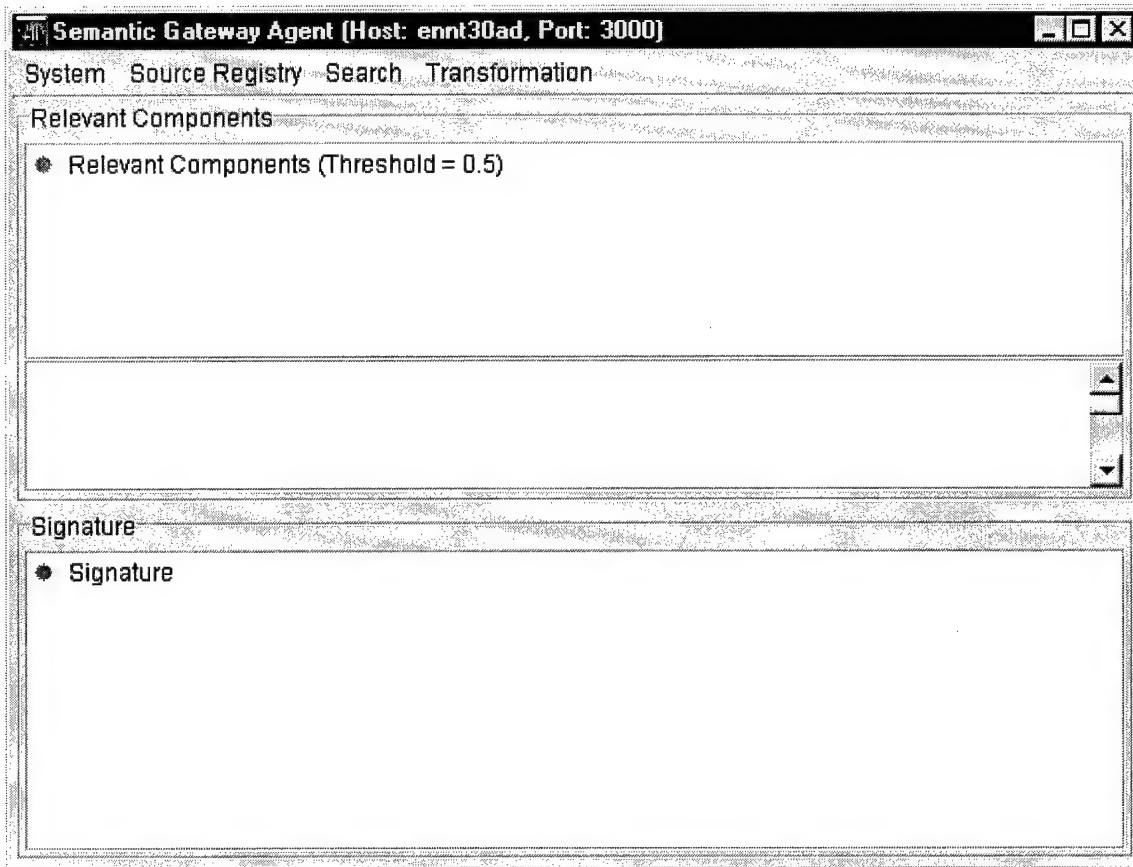


Figure 42: SemanticGateway Application

The *SemanticGateway* application requests and receives registry information from each *ScenarioRegistry* application, and also sends signature-based queries to these applications. To facilitate the retrieval of these data, the *SemanticGateway* creates an instance of a *SemanticGatewayAgent* object and executes it in a separate thread.

4.3.2.1 *SemanticGatewayAgent* Class

The class diagram of the *SemanticGatewayAgent* is shown in Figure 43. The diagram shows the *AFIT Agent MOM* base *Agent* and *Conversation* classes the *SemanticGatewayAgent* and *registryRequestConv_I* classes extend. Since there may be multiple *ScenarioRegistryAgents*, distributed across multiple machines, the *SemanticGatewayAgent* must create a separate conversation with each agent to request

its results. The *SemanticGatewayAgent* must keep track of all these conversations, so it knows when all responses have been received.

To deal with this problem, the responsibility of tracking conversation progress has been delegated to the *SemanticProcess* class. When the *SemanticGateway* calls the *retrieveRelevantComponents* or *getRegistry* method of the *SemanticGatewayAgent* a *SemanticProcess* object is created with the appropriate *type* attribute value (*getRelevant* or *sendRegistry*) and a unique *processNo* attribute value. As new *registryRequestConv_I* objects are created, they are added to the *SemanticProcess* object's *conversations* vector. As each of these conversations receives its reply, it calls the *addToResultSet* method of its *process* attribute object with a vector containing its results as the parameter. The *SemanticProcess* object's *addToResultSet* method adds the contents of the input vector to its *resultSet* attribute, decrements its *activeConvCount* attribute by one, and checks this attribute to determine if its value is less than one. If this test passes, all replies have been received and the *processComplete* method of its parent object is called with itself as the parameter. The *SemanticGatewayAgent* object's *processComplete* method calls the appropriate method of its parent, based on the value of the *SemanticProcess* parameter's *type* attribute, and passes the *SemanticProcess* object's *resultSet* as the argument. The *SemanticProcess* object is then deleted from the *processes* attribute of the *SemanticGatewayAgent*.

The *SemanticGatewayAgent* initiates conversations with *ScenarioRegistryAgent* objects to request registry data and signature queries, then collects the result sets from those agent's replies. Since there may be multiple *ScenarioRegistryAgents* distributed across several systems, the *SemanticGateway* application provides the capability to register and track these agents. This is accomplished through the source registry.

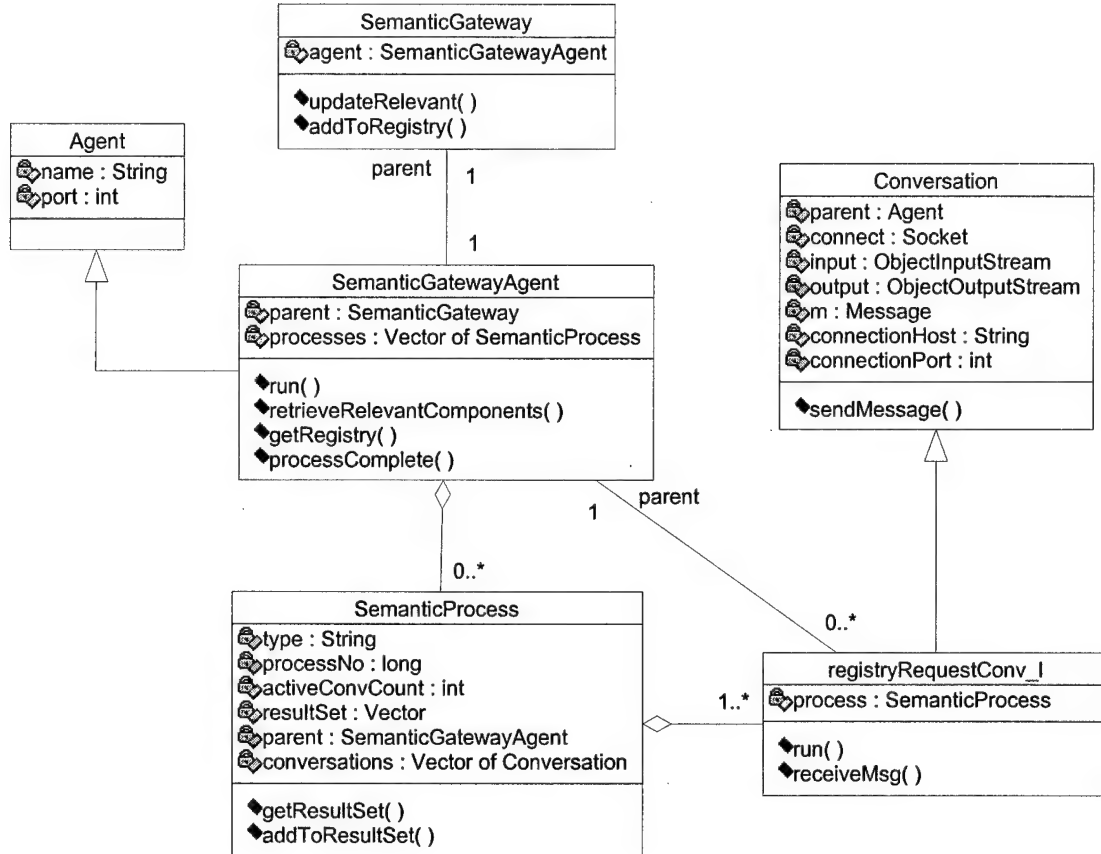


Figure 43: SemanticGatewayAgent Class Diagram

4.3.2.2 Source Registry

In the Semantic Broker architecture, the source registry contains all information required by the broker to access metadata and data files needed to generate scenario component object models, search for relevant components, and transform selected components to a target format. This includes keeping track of available *ScenarioRegistryAgent* objects.

The functionality of the source registry is contained in the *SourceRegistryGUI* and *SourceRegistry* classes. *SourceRegistryGUI* objects present the user with a means of reviewing and updating the contents of the system-wide source registry. *SourceRegistry*

objects are created by the *SemanticGateway* parent object during its initialization to instantiate the source registry data structure stored format. Figure 44 shows the classes that are part of the source registry portion of the *SemanticGateway* application.

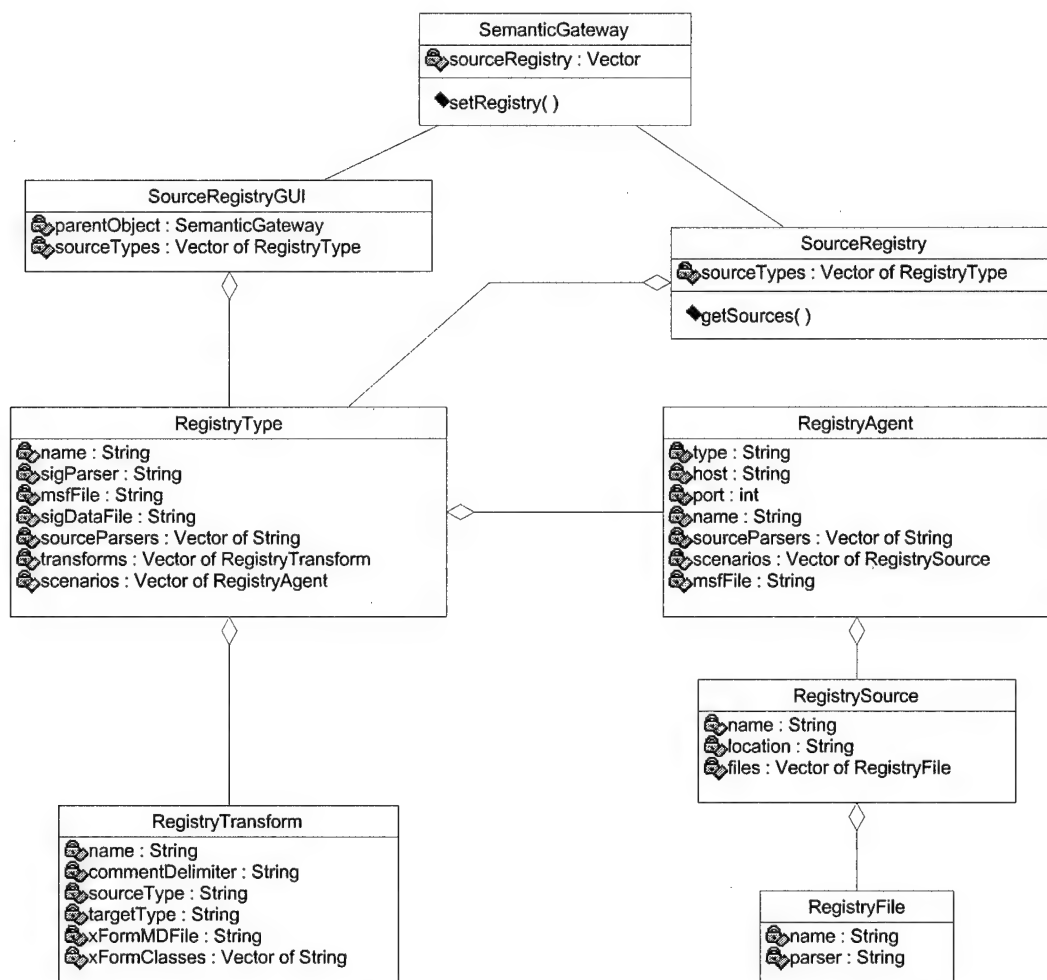


Figure 44: Source Registry Class Diagram

The *SourceRegistryGUI* and *SourceRegistry* classes each contain a Vector of *RegistryType* objects. A *RegistryType* object represents a simulator type (e.g. SUPPRESSOR) and its attributes contain the file names for signature data and metadata, as well as a string representing the class name for the signature *Parser* sub-class. Java's reflection mechanisms are employed to instantiate the applicable *Parser*

object from its string representation. Additionally, a *RegistryType* object contains a vector of *RegistryTransform* objects and a vector of *RegistryAgent* objects.

RegistryTransform objects represent a particular transformation process (e.g. SUPPRESSOR-to-SWEG). *RegistryTransform* objects contain *sourceType* and *targetType* attributes, a string representing the name of the file that contains the transformation metadata, and a vector of *String* objects that contains the names of the transform classes. As with the parser in the *RegistryType* object, Java's reflection tools are utilized to instantiate the applicable transforms from the *String* objects in the *xFormClasses* vector attribute.

The *RegistryAgent*, *RegistrySource*, and *RegistryFile* classes are discussed in detail in Section 4.3.1.2. A *RegistryType* object's *scenarios* attribute contains a *RegistryAgent* object for each *ScenarioRegistryAgent* referenced in the system-wide source registry and a *RegistryTransform* object for each transformation (e.g., SUPPRESSOR-to-SWEG) available for the applicable scenario type.

Figure 45 the Graphical User Interface (GUI) for the source registry. The GUI utilizes a Java *JTree* to graphically represent the organization of the source registry data.

In the figure, the letters provide a cross-reference between the *JTree* nodes in the figure and the underlying data structure presented in Figure 44.

- A. **RegistryType:** This object registers SUPPRESSOR as a source type for scenario components. After the name of the source type, the GUI displays the number of scenario agents that have been registered for that source type. The tree displays the class name of the signature parser, and the file names of the signature file and the syntax metadata file. It is important to note here that the signature file listed in this tree is accessed when adding new signatures to the SCDB. This file is text based and must be parsed and converted to object form.
- B. **RegistryTransform:** This object registers the SUPPRESSOR – SWEG transformation capability. The source and target type attributes are listed as well as the filename of the transformation metadata file and the comment delimiter for

the target type scenario format. The comment delimiter is used where a component, or portion thereof, cannot be transformed to the target format. In these cases, the name of the untranslatable source component is commented to notify the user of the transformation anomaly.

- C. This node's children are the names of the *Transform* classes for this transformation capability. See Section 4.3.5 for details on how transformations are accomplished in the Semantic Broker system.
- D. **RegistryAgent:** This object registers a SUPPRESSOR *ScenarioRegistryAgent* located on host *ennto5li*, port 2500. Each time the *SemanticGateway* application is started or the source registry is updated, the registry will request an update from each registered *ScenarioRegistryAgent* to determine whether each is operational. Registry entries for *RegistryAgent* objects that reference *ScenarioRegistryAgent* objects that are unavailable will have *UNAVAILABLE* displayed after their host name and port number.

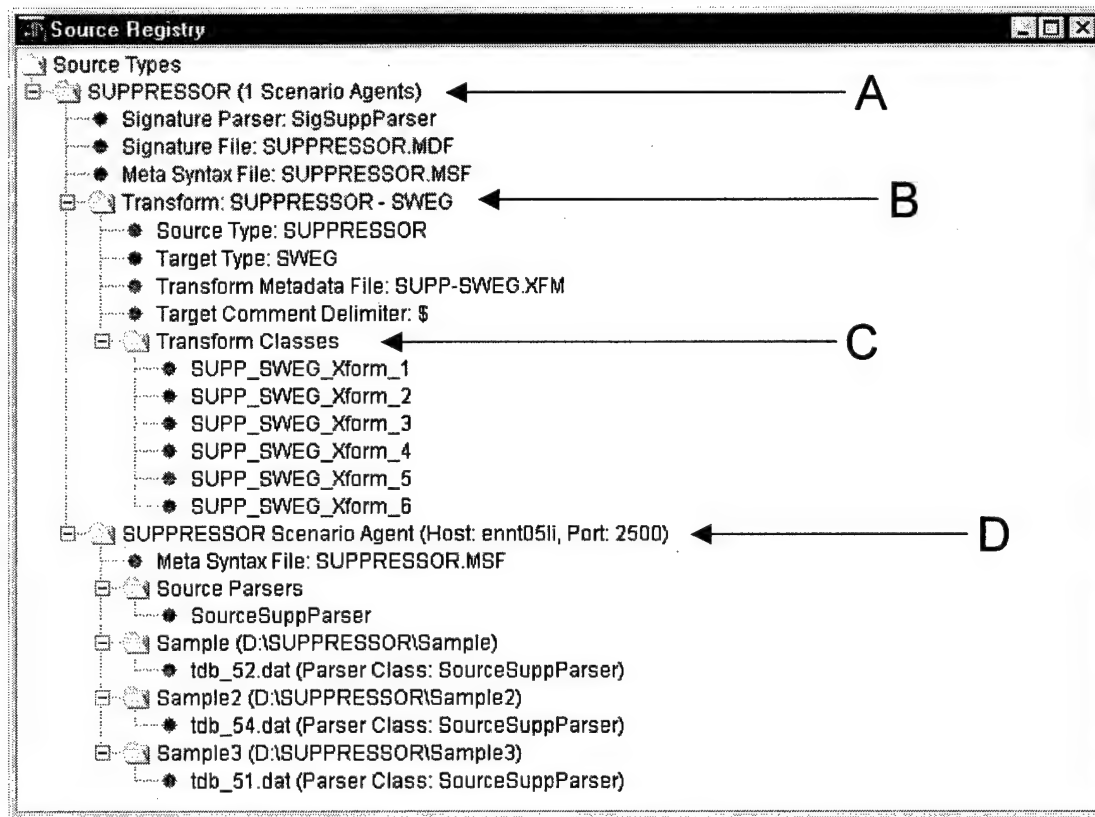


Figure 45: Source Registry Graphical User Interface

The source registry allows the user to configure the Semantic Broker to recognize the available scenario types, sources and files, and the necessary *Parser* and *Transform*

sub-classes. Once the source registry has been configured, a signature can be used to retrieve relevant components.

4.3.2.3 Component Retrieval

Component retrieval is one of the core functions of the *SemanticGateway* application. Component retrieval provides the user with the capability to identify existing scenario components for reuse in a simulation scenario currently under development. Since, under the scheme developed in this research, reusable components are identified based on their similarity to a signature component, component retrieval begins with signature selection.

Signature Selection

The Semantic Broker maintains all signature components in the SCDB. Objects of the *SignatureSelector* class provide the user with a GUI that allows the modification, deletion, and selection of signature components. The *Signature Selector* window is accessed via the *Search* menu of the *SemanticGateway* application.

The *SignatureSelector* window is shown in Figure 46. The *SignatureSelector* GUI uses a *JTree* to display the hierarchical structure of the signature components. Selecting a signature component with the mouse causes the comments for that signature to be displayed in the lower pane of the window. These comments can be updated by the user to enhance future user's understanding of the signature's capabilities and limitations. This is accomplished by updating the text in the *Comments for Selected Signature* text area and clicking the *Update Comments* button.

Modification and deletion of existing sub-components, as well as the addition of new sub-components and characteristics, is possible via a popup menu displayed when the right mouse button is depressed and released.

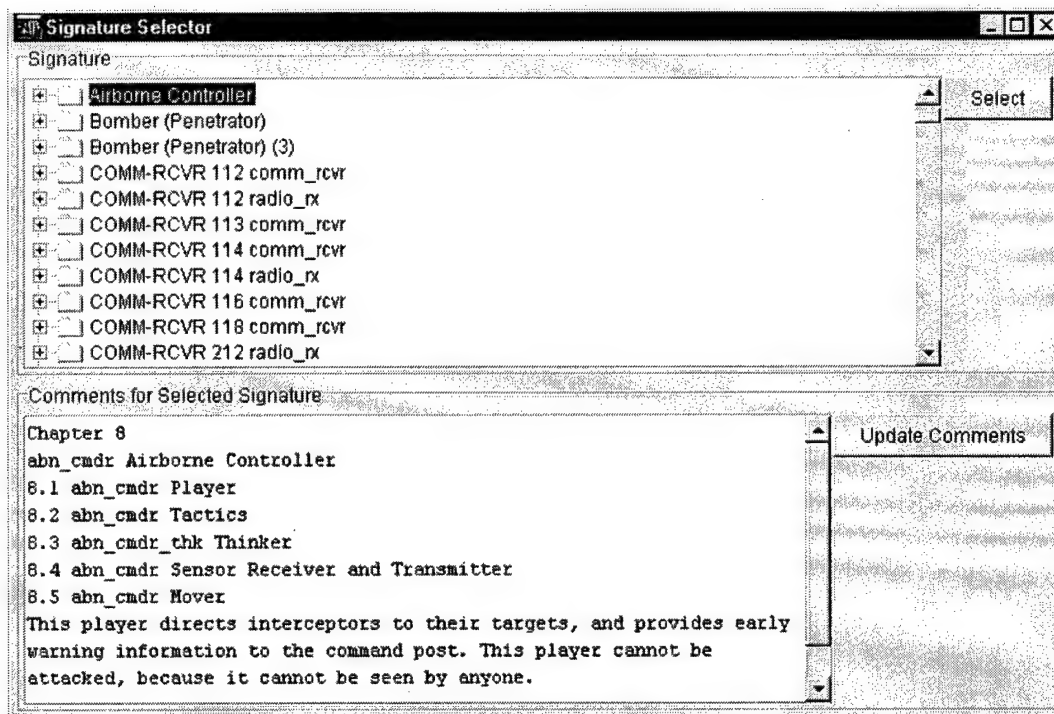


Figure 46: SignatureSelector Window

Figure 47 shows the *SignatureSelector* object's popup menu. The *Mandatory* and *Optional* menu items allow the user to tell the search engine whether a particular sub-component is absolutely essential for a source component to be included in the set of relevant components compiled during the search. The *Add*, *Edit*, and *Delete* menu items are self-explanatory, and the *Select* menu item sets the currently selected component as the search signature that will be used by the SemanticGateway application for relevant component retrievals. The *Select* button in the upper right-hand portion of the window serves the same function as the *Select* menu item. Selecting a signature component via either method causes the *SignatureSelector* object to display an information dialog box verifying the users signature selection.

The *SignatureSelector* is terminated by closing the window. This action updates the SCDB to reflect any changes accomplished by the user. After a signature has been selected, relevant components can be retrieved from the available scenario sources.

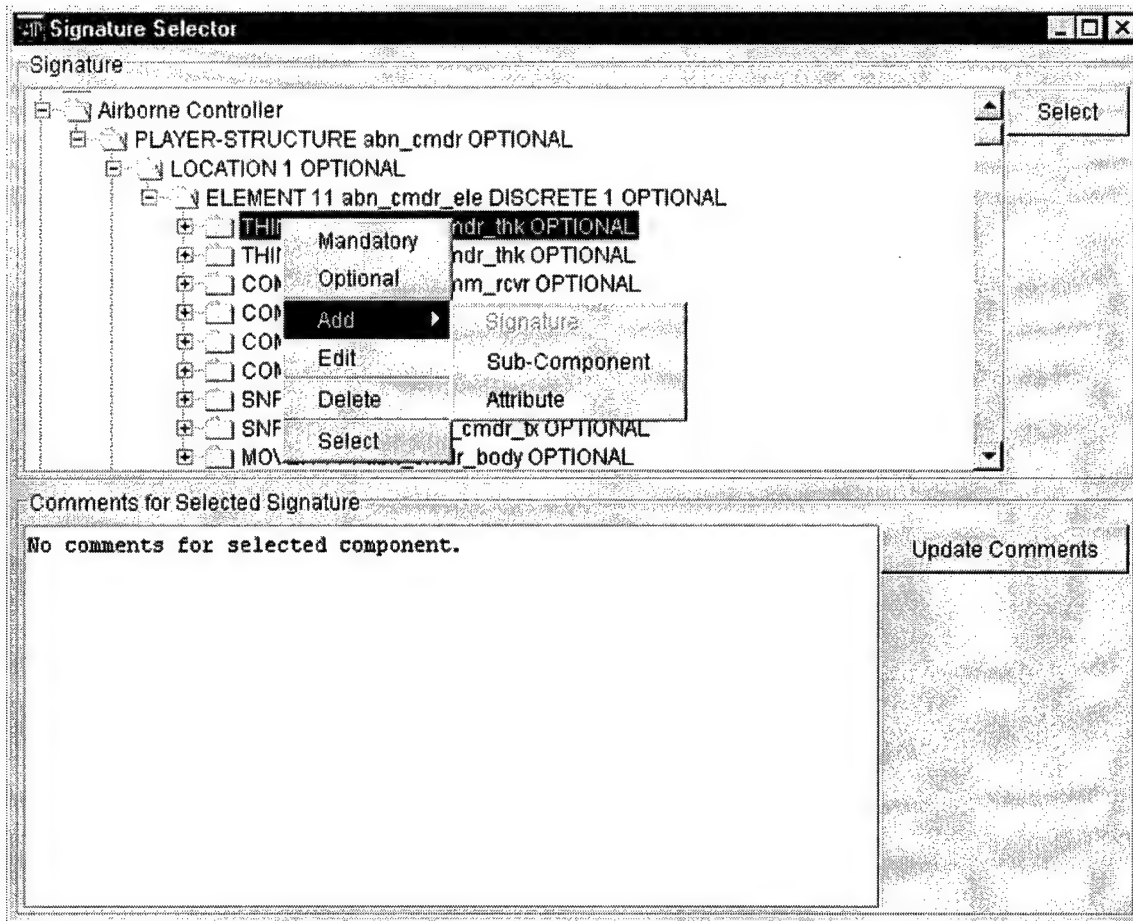


Figure 47: SignatureSelector Popup Menu

Relevant Component Retrieval

Source component analysis is performed by measuring a source component's degree of similarity to the selected signature component. Essentially, the signature component is a query structure, and source components are scored on how well they meet the query criteria. The subject of signature analysis was covered in Section 4.2.3.

Figure 48 shows the *SemanticGateway* window after the signature selection process has been completed. The selected signature component, an *Airborne Controller*, is displayed in the lower half of the window. To retrieve relevant source components, the user selects *Retrieve Relevant Components* from the *Search* menu.

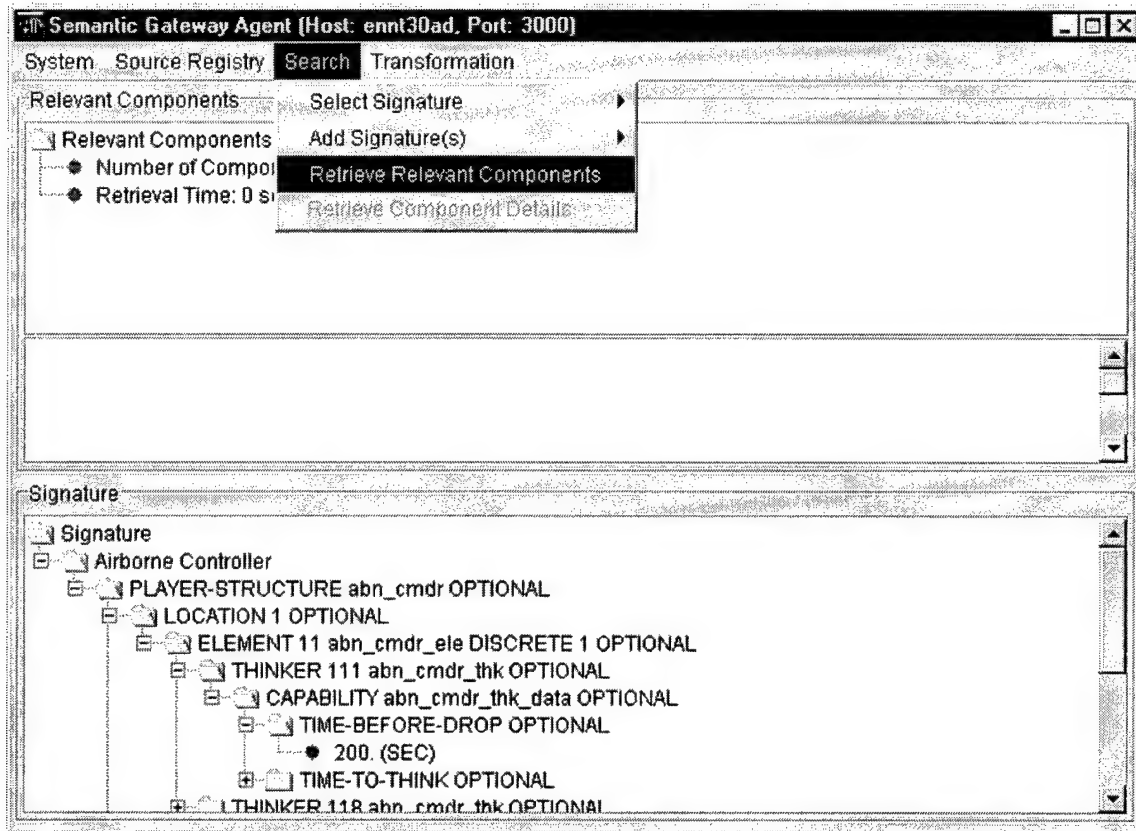


Figure 48: SemanticGateway Application with Signature Selected

This action causes the *SemanticGateway* object to call the *retrieveRelevantComponents* method of its *SemanticGatewayAgent* object. The *SemanticGatewayAgent* accesses the source registry's collection of available *RegistryAgent* objects and initiates a conversation with each referenced agent. Each agent queries its collection of scenario source files, and returns the resulting set of relevant component references to the *Conversation* object that initiated the conversation. As each conversation terminates, its results are included in the overall result set. After all conversations have terminated, the *SemanticGatewayAgent* calls the *updateRelevant* method of its parent *SemanticGateway* object. This effectively updates the list of relevant component references displayed in the upper half of the *SemanticGateway* window. The resulting updated *SemanticGateway* window is shown in Figure 49.

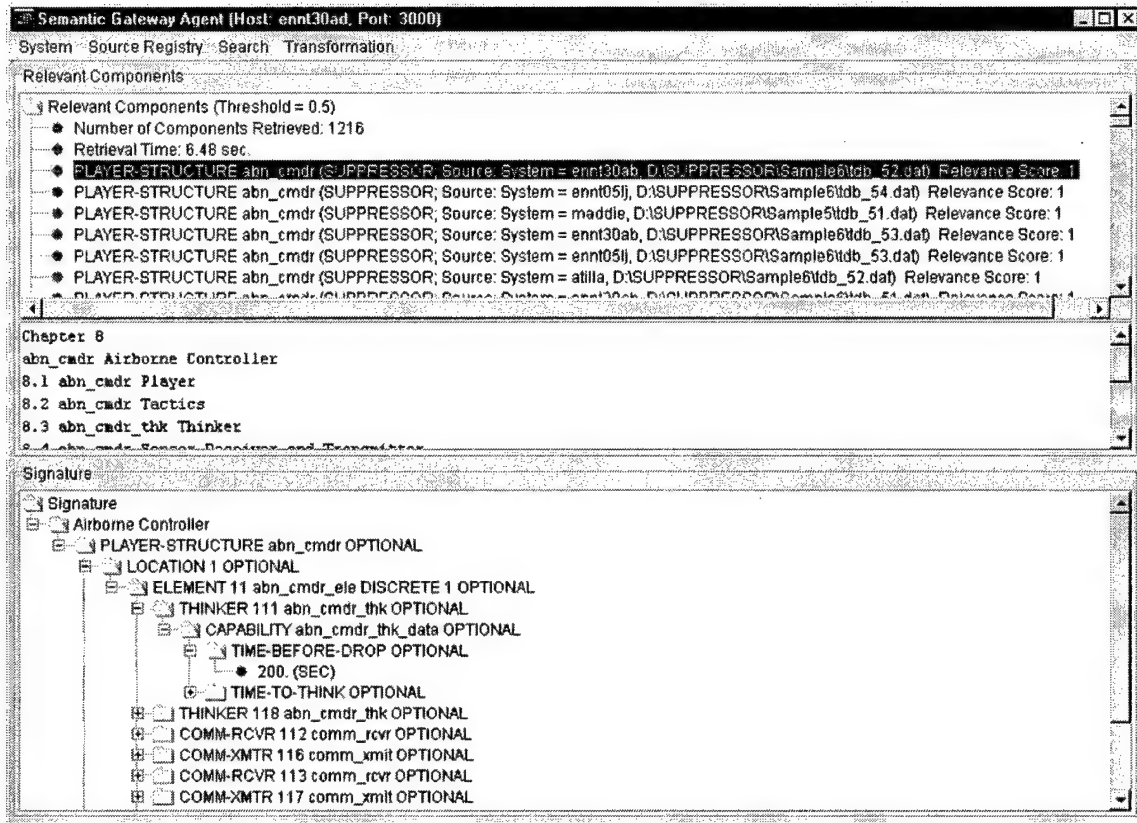


Figure 49: SemanticGateway Window After Relevant Component Retrieval Process

The relevant component references are displayed in the upper half of the window, and the comments associated with the selected relevant component reference are displayed in the text area below the relevant component pane. Here again, a *JTree* is used to display the hierarchical structure of the relevant source components. The relevant component references are sorted in descending order based on their relevance score. As discussed in Section 4.2.3, a relevance score of '1' indicates that the source component contains the entire structure and characteristics of the signature exactly.

By selecting a relevant component reference, clicking the right mouse button, and selecting *Retrieve Component Details* from the popup menu; the user directs the *SemanticGateway* to retrieve the entire object model for the selected reference. This menu is shown in Figure 50. The *SemanticGateway* retrieves the object model by

calling the *getCompDetails* method of its *SemanticGatewayAgent* object. The *SemanticGatewayAgent* object creates a *registryRequestConv_1* object and passes it a Message object with attribute *performative* equal to *getCompDetails* and attribute *content* equal to the relevant component reference. The *registryRequestConv_1* object then contacts the *ScenarioRegistry* application indicated by the relevant component reference.

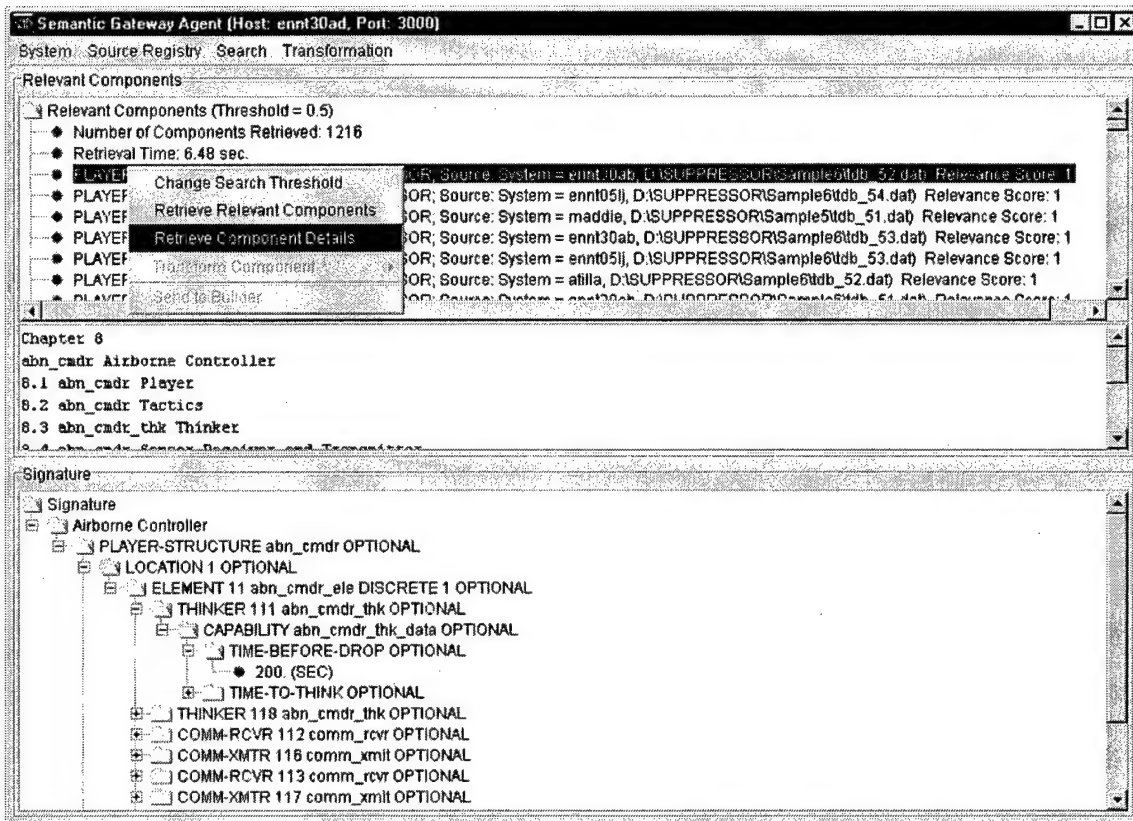


Figure 50: SemanticGateway: Retrieve Component Details Menu

After the *ScenarioRegistry* application responds with the object model, the model's nodes are added to the appropriate relevant component reference node. The user can then expand the relevant component and examine its sub-components to determine if it is suitable for reuse in a new scenario. Figure 51 shows the *SemanticGateway* window after the details for the selected relevant component reference have been retrieved and

the *JTree* updated. Once selected for reuse, a component formatted for a different simulator type than required must be transformed to the desired format.

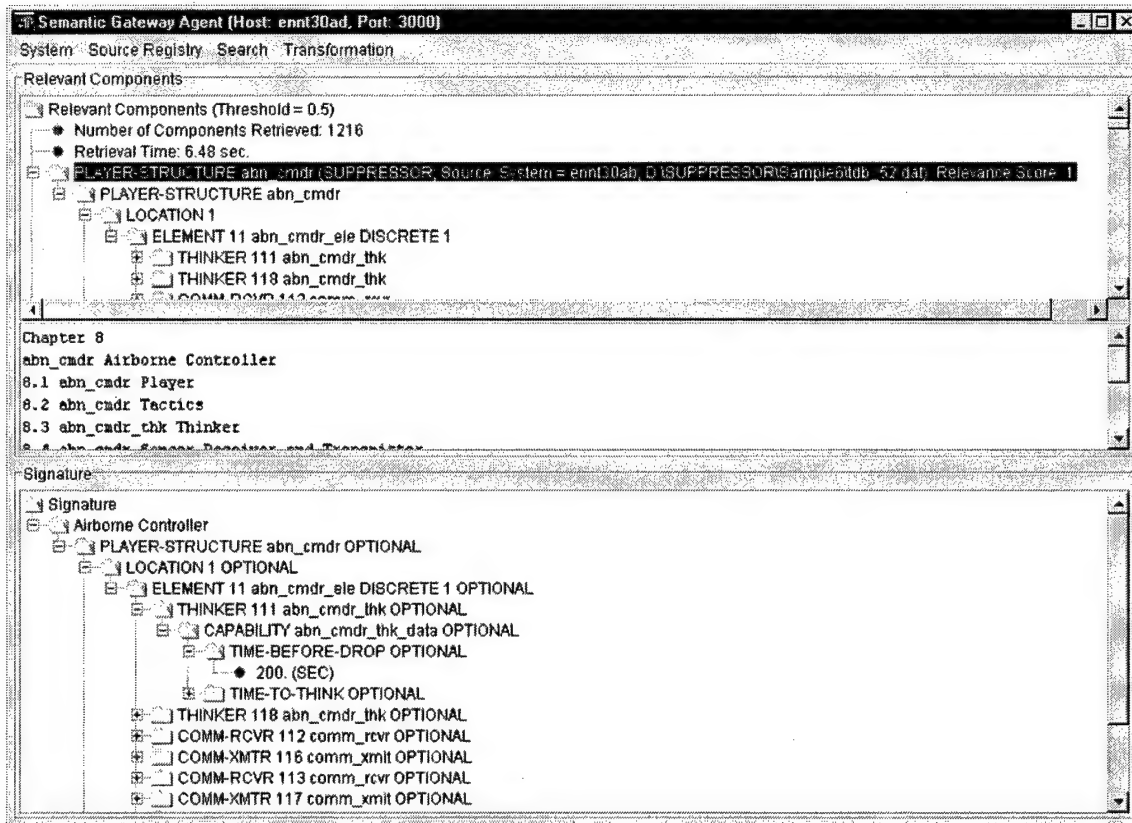


Figure 51: SemanticGateway: Component Details Expanded

4.3.2.4 Component Transformation

The *SemanticGateway* divides transformations into six categories [LSA98]. These were discussed in detail in Section 3.5.2.4. Transformations are performed by an instance of the *TransformEngine* class. Transformation metadata is utilized to determine which class of transform object to use to translate a specific component or sub-component. References to this metadata and the transform classes used during the transformation process are contained in the source registry. The use of transformation metadata allows the methods of the *TransformEngine* class to be generic, and the division of the transforms into categories permits large components to be translated one

sub-component at a time. This avoids the problem of one untranslatable sub-component rendering the entire component untranslatable. The specifics of the transformation process are moved into the transform classes. This feature makes the system more easily extendable, since none of the core source code requires modification to add a new transformation (e.g. SUPPRESSOR – SWEG, SUPPRESSOR – JIMM, etc.).

In Chapter 3 of this work, a class diagram is described for the proposed design of the transformation portion of the *SemanticGateway* application. This class diagram is shown in Section 3.4.2.3, Figure 32. A more detailed diagram that shows the actual implementation of the classes of the transformation sub-system is shown in Figure 52. In the Semantic Broker architecture, all transform classes must extend the *Transform* class. This research developed the *Transform* sub-classes necessary to translate SUPPRESSOR scenario components to SWEG components. The abstract class *Transform* contains the abstract method *transform*, and each class that extends it must provide its implementation of this method.

The Semantic Broker architecture requires that a transform class be provided for each of the six transform categories detailed in Section 3.3.4, Figure 26. These classes are referenced in the source registry via *String* objects that contain their class names. Reflection is utilized to instantiate objects from the registry's string reference. There is no requirement that the transforms for the six categories be unique, so, for example, the same transform class name could be provided for both Category 5 and Category 6 transformations.

The *SemanticGateway* application performs transformations by creating a *TransformEngine* object and passing to it the appropriate *RegistryTransform* object. The constructor of the *TransformEngine* class performs the following functions:

- Extracts the source and target types, the *Transform* class string references, and the transformation metadata filename from the *RegistryTransform* object.
- Instantiates the *Transform* sub-classes referenced in the *RegistryTransform* object by calling its *getTransforms* method. This method utilizes Java's reflection tools to instantiate *Transform* objects from the strings stored in the *xFormClasses* attribute of the *RegistryTransform* object.
- Creates an *MDParser* object for the metadata file referenced by the *RegistryTransform* object.
- Calls the *loadXFormMetadata* method to initialize the *xFormMD* attribute with the transformation metadata.

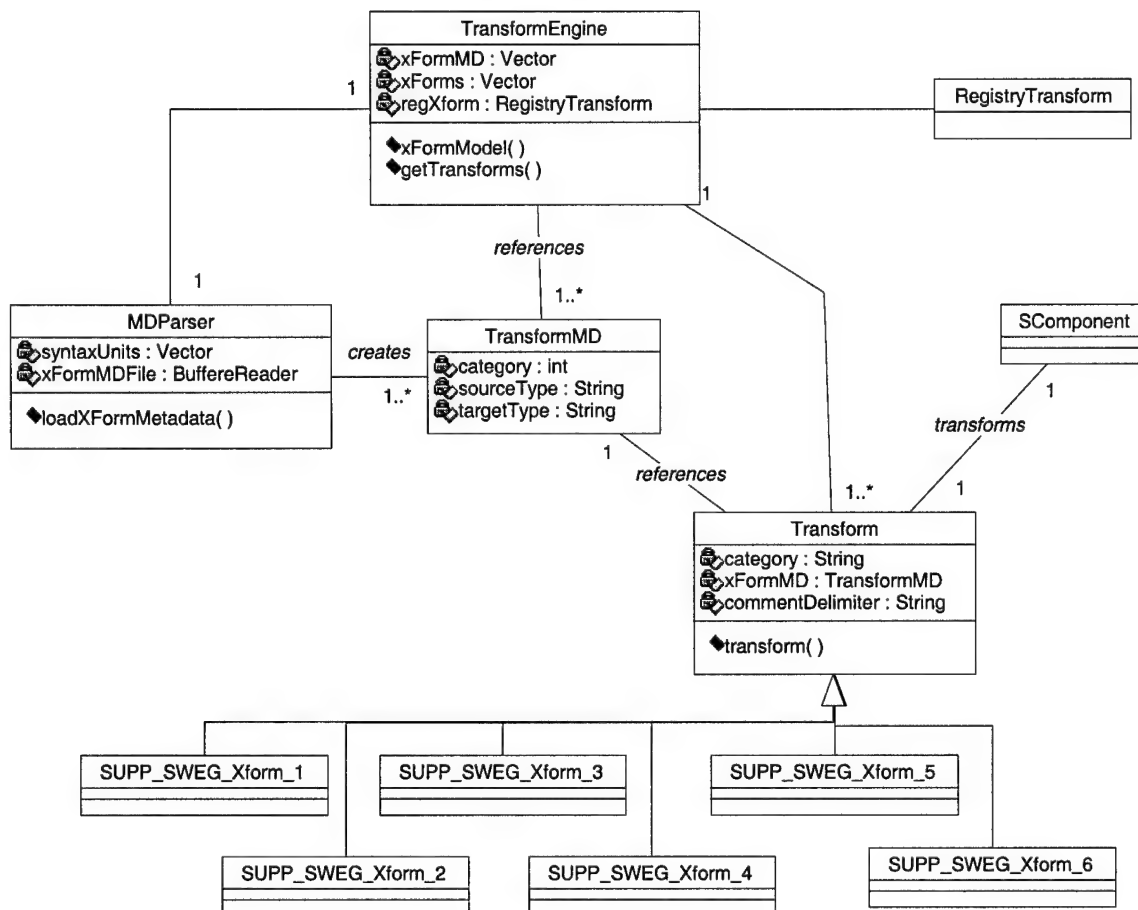


Figure 52: Transformation Sub-System Class Diagram

The *SemanticGateway* application then calls the *TransformEngine* object's *xFormModel* method and passes the root object of the selected scenario source component as the argument. This method calls the *xFormComp* method, which

transforms the *SComponent* object passed as the argument to the *xFormModel* method.

Figure 53 contains the source code for the *xFormModel* and *xFormComp* methods.

```
public SComponent xFormModel(SComponent o)
{
    this.sourceComp = o;
    SComponent rtn = null;

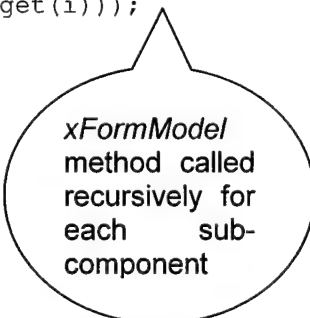
    // Transform input parameter 'o'
    rtn = xFormComp(o);

    // Transform the sub-components of input parameter 'o'
    // and add them to the transformed component.
    int i;
    Vector comps = o.getComponents();
    for (i = 0; i < comps.size(); i++)
        rtn.addComponent(xFormModel((SComponent)comps.get(i)));
    return rtn;
}

public SComponent xFormComp(SComponent o)
{
    SComponent rtn;
    // Find transform metadata object. If not found
    // abort component transformation.
    TransformMD xformMD = findXFormMD(o.getType());
    if (xformMD == null)
        return null;

    // Retrieve Transform sub-class object based on category
    // data contained in transformation metadata object. If
    // transform is null abort.
    Transform xform = (Transform)this.xForms.
        get(xformMD.getCategory() - 1);
    if (xform == null)
        return null;

    // Set transform metadata, then call Transform object's
    // transform method with source component as parameter.
    xform.setXFormMD(xformMD);
    rtn = xform.transform(o);
    return rtn;
}
```



xFormModel
method called
recursively for
each sub-
component

Figure 53: TransformEngine Class *xFormModel* and *xFormComp* Methods

The *xFormComp* method translates the name of the component and its characteristics. The sub-components of the argument are ignored by this method. The sub-components are each treated as the root of another scenario component object

model. Therefore, for each sub-component, the *xFormModel* method is called recursively with the sub-component's root object as the parameter. This design feature allows a portion of a scenario component to be untranslatable without rendering the entire component untranslatable. A component that cannot be transformed is simply annotated as such, and the process continues with its sub-components. As can be discerned from closely examining the source code of the *xFormModel* method, the component parameter is translated first using the *xFormComp* method, then the *xFormModel* method is called recursively for each of the sub-components. This line in the source code of Figure 51 is identified by the callout. This process continues until the leaf nodes of the source component have been reached and transformed.

Figure 54 provides a graphical representation of the how these two methods are employed to transform scenario components. As the figure shows, the non-*SComponent* attributes of the *SComponent* object passed to the *xFormModel* method are transformed by the *xFormComp* method. The *SComponent* attributes, those contained in the *components* attribute, are transformed individually by recursively calling *xFormModel* *n* times, where *n* is the number of *SComponent* objects in the "components" Vector attribute of the current *SComponent* object.

In the figure, the *SComponent* objects are labeled (i.e. A, A.1, A.2, A.2.1, ...) to indicate their hierarchical relationship. Only the root of the transformed component is shown (with 'T' suffix). Of course, the object model shown in Figure 54 is very small and the object representation of almost any scenario component is many times more complex. After the leaf nodes of the component object model have been transformed, and the recursive calls have terminated, the original call to *xFormModel* returns the root of the transformed component to the *SemanticGateway* application.

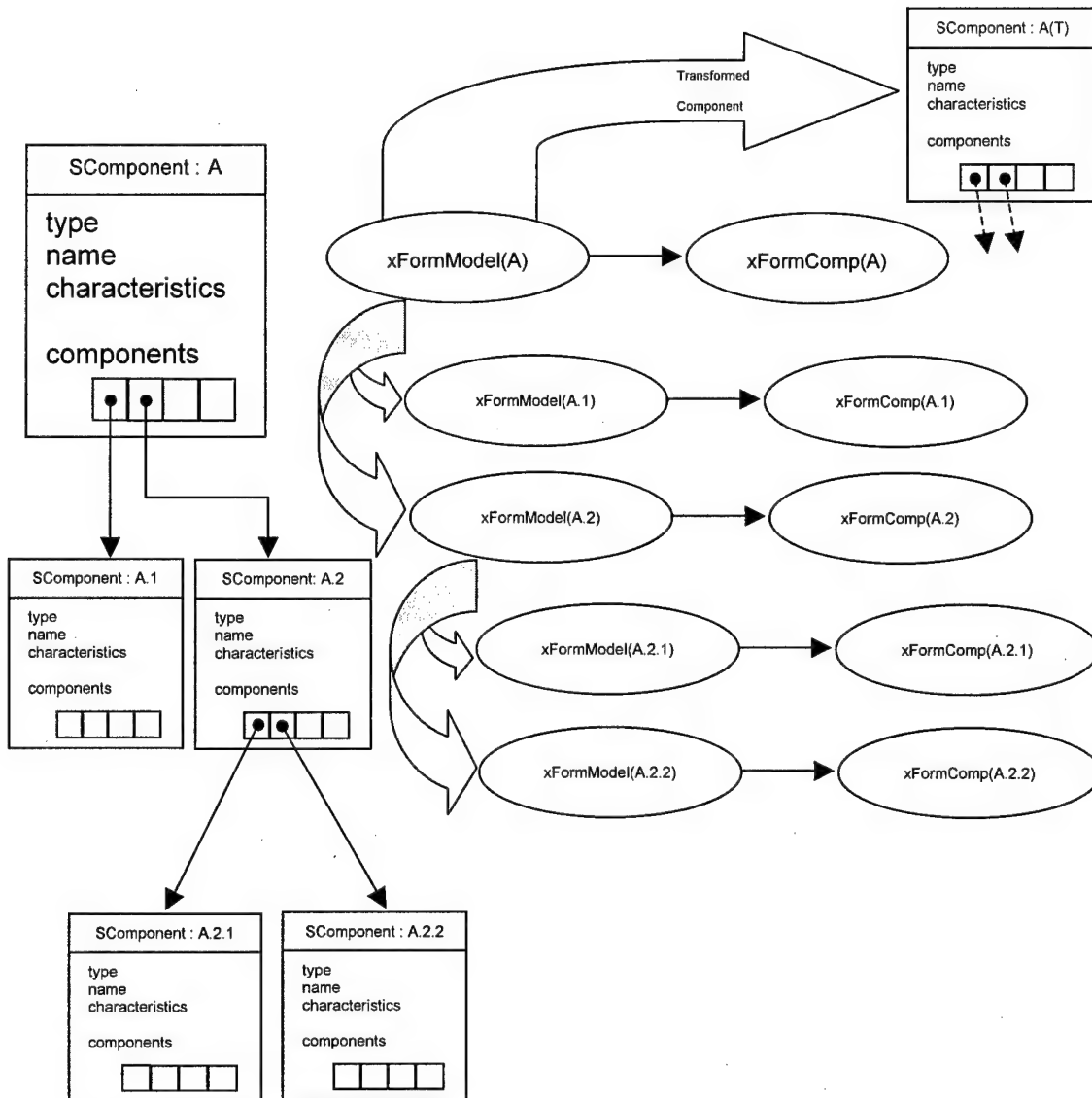


Figure 54: Component Transformation Process

To transform a scenario component, the user first completes the signature selection and relevant component retrieval processes. Once this is done, the user selects a component from the relevant component list, then selects a transformation option from those listed in the *Transform Component* sub-menu of the *Transformation* menu. Figure 55 shows the *SemanticGateway* window with this popup menu displayed.

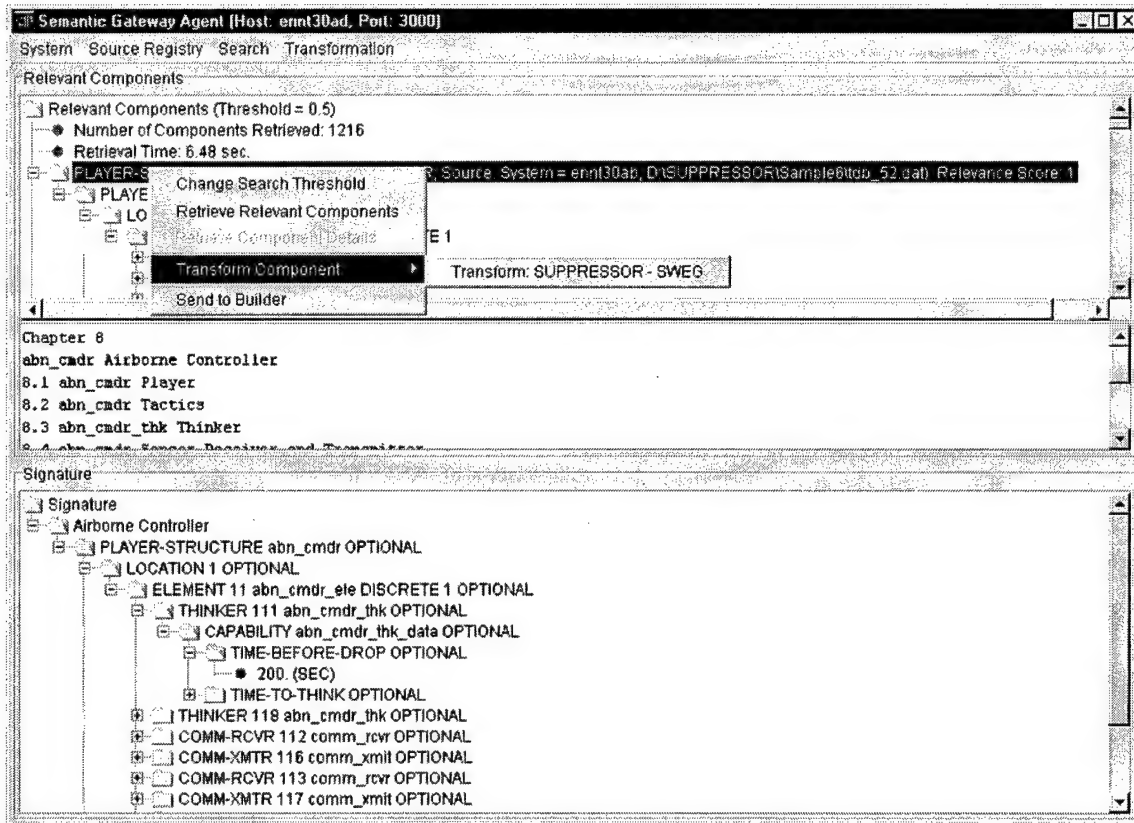


Figure 55: Transformation Menu of the SemanticGateway

In the figure, there is only one transformation option listed. This is because the *Transform Component* sub-menu items are generated automatically based on the available *RegistryTransform* objects in the source registry. By selecting the *SUPPRESSOR – SWEG* transformation option, the user causes the *SemanticGateway* application to execute its *transformComponent* method, which effectively translates the selected component to the target format (i.e., SWEG). The resulting transformed component is displayed in an instance of the *ComponentViewer* class. The *ComponentViewer* class provides a GUI that allows the user to inspect the transformed component. The *ComponentViewer* window with the transformed SUPPRESSOR scenario component is shown in Figure 56.

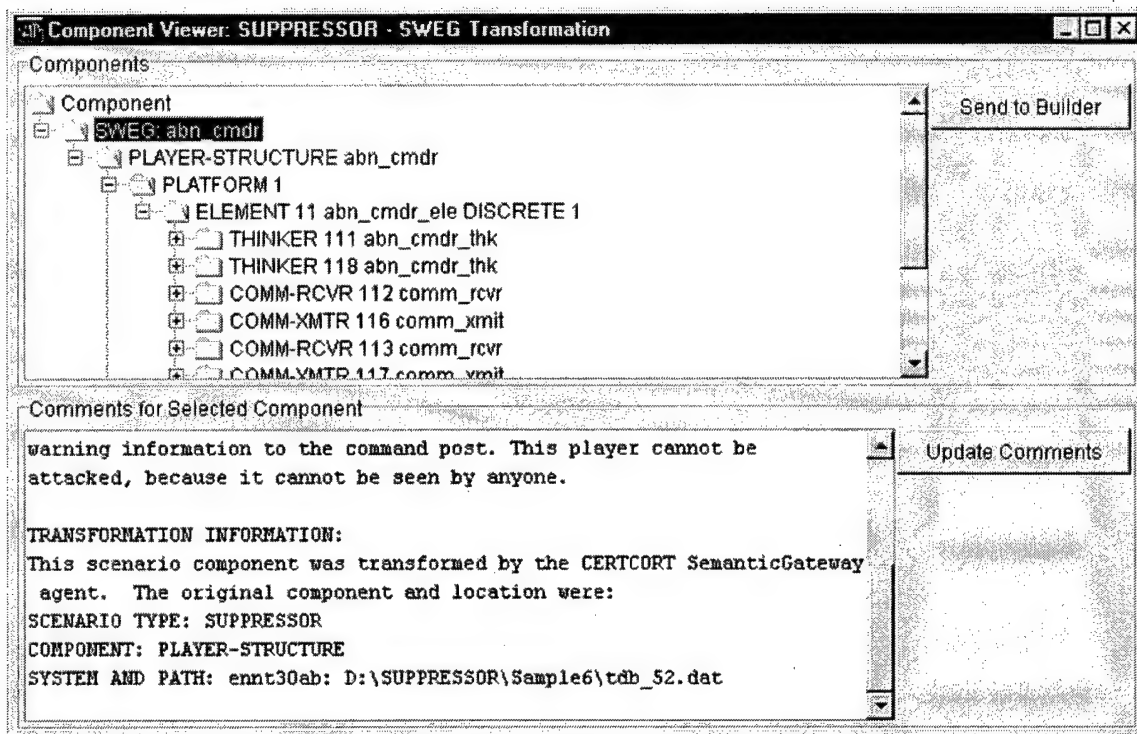


Figure 56: ComponentViewer Window

As shown in the comments pane of the *ComponentViewer* window in Figure 56, the *SemanticGateway* application's *transformComponent* method annotates the comments of the transformed component to ensure that future users of the component will know the original scenario format and the scenario file from which it was transformed.

The design of the transformation sub-system allows sub-components to be untranslatable without rendering the entire component unusable. In Figure 57, the *JTree* has been expanded to show some of the TACTIC sub-components that could not be transformed because they are not available in the target format.

These items are identified by the "\$ ITEM NOT AVAILABLE IN TARGET FORMAT: <SOURCE TYPE>." The '\$' is the comment delimiter for the target format. This item is configurable in the source registry by editing the associated *RegistryTransform* object. Annotating the untranslatable sub-components allows the user to easily identify where

transformation problems occurred and which sub-components must be manually translated.

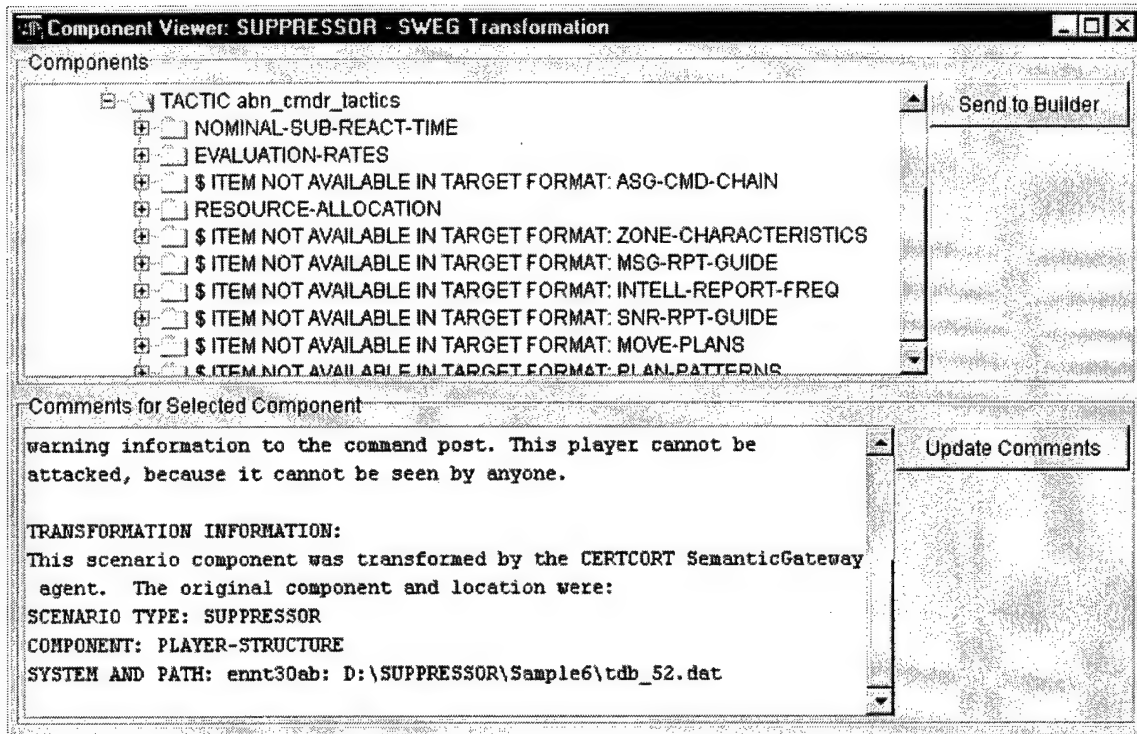


Figure 57: Transformed Component with Untranslatable Components

Since the goal of this research did not include developing a repository for transformed components or developing facilities to construct new scenarios from the transformed components, transformed components are discarded when the *ComponentViewer* window is closed.

4.3.3 SemanticGateway – ScenarioRegistry Interaction

The purpose of this section is to give the reader a better understanding of the interaction between the *SemanticGateway* application and the *ScenarioRegistry* applications. Figure 58 shows the primary classes of each application.

The figure shows the applications, their agents, and the conversation objects that communicate between them. Network communication via Java *Socket* objects is

depicted by dashed lines. In the figure there is only one *ScenarioRegistry* application, so only one *registryRequestConv_I* object was created by the *SemanticGatewayAgent* to contact it. In practice there could be multiple *ScenarioRegistry* applications. In these cases a separate *registryRequestConv_I* object would be created to connect with and request information from each *ScenarioRegistryAgent*.

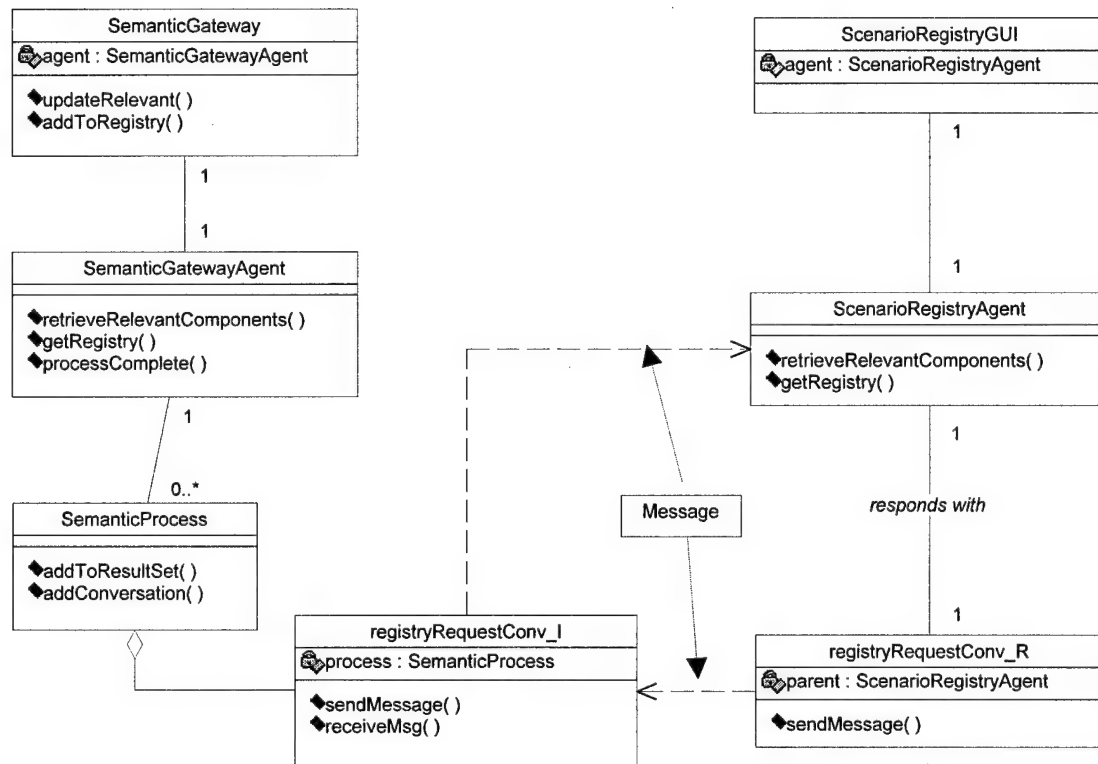


Figure 58: SemanticGateway – ScenarioRegistry Interaction

4.4 Semantic Broker Demonstration

This section begins with a discussion of the supporting software required to compile and run the semantic broker system. Next the virtual machine, compiler, and hardware platforms are discussed. Finally, the software is configured and tested to determine relevant component retrieval feasibility in operational conditions.

4.4.1 Java Packages

In order to compile and execute the semantic broker software, the *afit.mom* package must be available. This package contains the base classes for the AFIT Agent MOM multi-agent development API. The location of this package is critical. For example, if the semantic broker software is in the *C:\CERTCORT\SemanticBroker* directory, the *afit.mom* class files must be in the *C:\CERTCORT\SemanticBroker\afit\mom* directory. All other packages imported by the semantic broker software are part of the standard java packages delivered with Sun Microsystems' Java Development Kit 1.3 (JDK 1.3).

4.4.2 Hardware and Software Platforms

The semantic broker software has been developed in the Java programming language with the latest release (1.3) as its preferred runtime environment. Although developed and tested on the Windows NT/Intel platform, the portability of the Java language, with its platform independent Java Virtual Machine (JVM), makes the semantic broker capable of operation across heterogeneous platforms without modification of the source code.

4.4.3 Component Retrieval Test Cases

In order to determine the feasibility of using this component retrieval tool when reasonably large numbers of scenario source files are available, the *Scenario Component Database* for the test cases was constructed from 40 SUPPRESSOR scenario TDB files. Each of the 40 source files contained 21 PLAYER-STRUCTURE components, and the total size of the test database was approximately 10.7 *mega bytes* (MB) on each *ScenarioRegistry* machine. Two separate hardware configurations were employed during testing. The first tests the software on homogeneous platforms, and the second compares retrieval times across heterogeneous systems.

4.4.3.1 Homogeneous Test Configuration

Figure 59 shows the configuration of the hardware and software utilized for the homogeneous tests. All hardware platforms involved in this testing were Intel Pentium based workstations running Microsoft Windows NT. The semantic broker software was compiled under Sun Microsystems' JDK 1.3, and each workstation was running version 1.3 of the Java Virtual Machine (JVM). The tests utilized four separate signature components, and the component retrieval process was conducted five times for each signature—at a different sensitivity threshold each time.

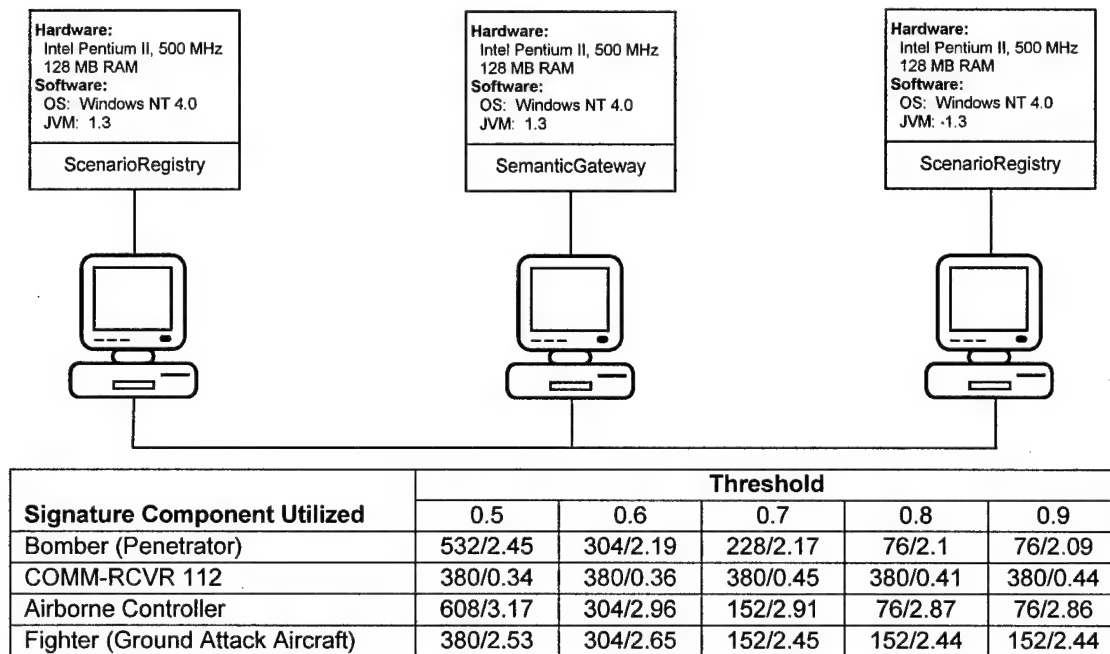


Figure 59: Homogeneous Test Configuration

This configuration utilized two workstations running the *ScenarioRegistry* application. Each of these applications accessed a *Scenario Component Database* of 10.7 MB. Therefore the total search space for these test cases was approximately 22 MB. Figure 59 also contains the results of the tests in the table in the lower portion of the figure. The left column lists the signature component utilized, and the remaining five

columns show the *number of components retrieved/retrieval time (sec)* for each of the sensitivity thresholds.

Contrary to what might be expected, the retrieval times were reasonably consistent regardless of the sensitivity threshold setting. The higher the threshold setting, the higher a source component's relevance score must be in order to be included in the relevant component set returned to the *SemanticGateway* application. At a higher threshold setting, there are fewer relevant component references returned by each *ScenarioRegistry* application. The heterogeneous tests showed similar results.

4.4.3.2 Heterogeneous Test Configuration

The second configuration tested the semantic broker software in a heterogeneous environment. This configuration is depicted in Figure 60.

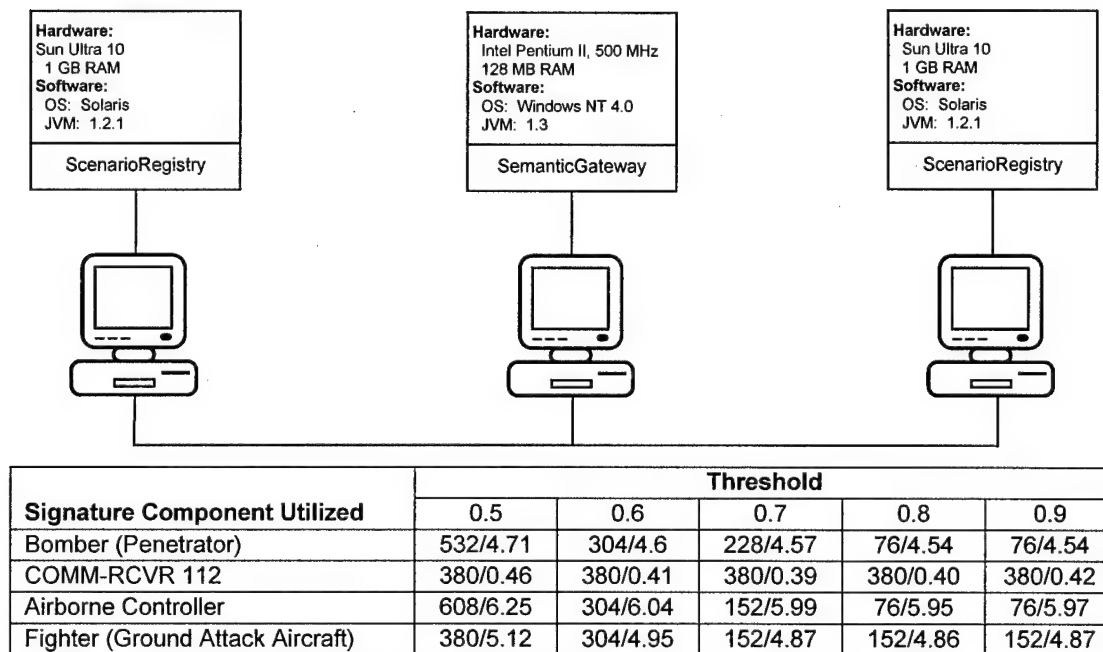


Figure 60: Heterogeneous Test Configuration

As the figure shows, this configuration employed two Sun Microsystems Ultra 10 workstations running the *ScenarioRegistry* applications. Each Sun workstation

contained 1 *giga byte* (GB) of memory. Version 1.2.1 of Sun Microsystems' JVM was installed on these machines. An Intel Pentium based workstation was utilized to run the *SemanticGateway* application. This machine had 128 MB of memory and version 1.3 of the JVM.

The size of the *Scenario Component Database* utilized for this test configuration was the same size as that used in the previous configuration. The response times of the heterogeneous configuration are comparable to that of the homogeneous configuration. The fact that the heterogeneous configuration's retrieval times are approximately twice that of the homogeneous configuration is most likely due to the conversions that are required when cross-platform communications are joined. Additionally, the Sun machines were running an older version (1.2.1) of the JVM, which may also contribute to their slower retrieval times.

Another interesting result of the tests was the time difference caused by signature components of different sizes. Since the component analysis process is controlled by the structure (i.e., the size) of the signature component, selection of a smaller signature component (e.g., COMM-RCVR) results in a faster component retrieval time.

4.5 Extending the Semantic Broker

The use of metadata, especially in the transformation portion of the semantic broker, increases the extendibility of the system. This section discusses the additional files and source code required to add new simulator scenario search and transformation capabilities.

When a new simulator scenario source is introduced, the system's source registry must be updated for the system to recognize the source files. This requires the addition of the following data files and Java *.class* files:

- **Signature Data File:** This file contains a text-based representation of the desired signature components for the new scenario type. The contents of this file will make up the initial signature database for the new type.
- **Syntax File:** This file tells the signature and source parsers how to interpret the signature and source files.
- **Signature Parser:** This is a Java *.class* file. It must extend the abstract class *Parser*.
- **Source Parser:** This is also a Java *.class* file. It must extend the abstract class *Parser*.

To add a new transformation capability to the system, the source registry must be updated with the associated data files and *Parser* class names, so the system recognizes the new transformation capability. The data files and Java *.class* files that must be added are:

- **Transformation Metadata File:** This file contains information concerning the transformation categories of all source components and sub-components. Characteristic transformations are included here, as well as additional characteristics that are present in the target format, but not supported in the source format.
- **Transformation Classes:** These are Java *.class* files. Each must extend the abstract class *Transform* and implement its abstract *transform* method.

The system needs a reference to six *Transform* classes. Any or all of these may be duplicates (i.e., reference the same class); however, in practice having only one transform class would not be effective.

No changes are required to the *SemanticGateway* application to update menu options because menus that can change due to added source types or transformation capabilities are generated from the content of the source registry at run-time. Therefore, updating the source registry to reflect an additional source type, for example, will automatically update the application's menus the next time it is started.

4.6 Summary

This chapter presents the implementation of the Semantic Broker as outlined in Chapter 3. The tool presented in this chapter is a proof of concept vehicle, and, as such, does not contain optimized data structures or algorithms that provide peak efficiency of space and execution time.

The chapter begins with a discussion of some design issues that came to light during development of the broker. These include the object model used to represent scenario components and generation of those object representations. Additionally, the topic of signature component analysis is covered to provide the reader with some level of understanding of its use in this research. Next, the implementation of the two main components of the broker, the *SemanticGateway* and *ScenarioRegistry* applications are discussed in detail. Following this was a short section on the results of tests conducted on the component retrieval algorithm to estimate its performance on reasonably large source databases. Finally, extending the tool to include new source types and transformation capabilities is covered.

Chapter 5, the final chapter of this work, follows. It provides conclusions arrived at as a result of this research, as well as some recommendations for future research in this area.

5. CONCLUSIONS AND RECOMMENDATIONS

5.1 Introduction

This chapter begins with a summary of the work conducted during the course of this research. This is followed by a discussion of the impact this research has had on the state of the CERTCORT system. Finally, several areas of future research that would further extend the capabilities of the CERTCORT system are discussed.

5.2 Summary of the Research

This research developed an agent-based system that provides users with an automated means of identifying existing scenario components and preparing them for reuse in a new scenario. Figure 61 shows how the tool developed in this work fits into the layered architecture of the CERTCORT multi-agent framework.

The Semantic Broker has two main components: the *SemanticGateway* application and the *ScenarioRegistry* application. The *ScenarioRegistry* application resides in the *Information Layer* of the framework, while the *SemanticGateway* application is positioned in the *Assistant Layer*. This places the *SemanticGateway* application in the same layer as the *Scenario Builder Assistant*, which has yet to be developed completely. Future research may determine that the *SemanticGateway* should be integrated into the *Scenario Builder Assistant*, since finding suitable existing scenario components and, if necessary, transforming them to the desired format, are key features of a *Scenario Builder Assistant*. This assistant is part of the original CERTCORT system as envisioned in [McD00, 209].

The signature analysis approach developed in Chapter 3 and implemented in Chapter 4 facilitates the identification of existing components, and the transformation

methodology developed in this work provides translation capabilities, within limits, between scenario formats. The key to both these functions is the simple object model utilized to represent scenario components. Its use permits the representation of virtually any scenario format and places the scenario components in a simulator system independent format.

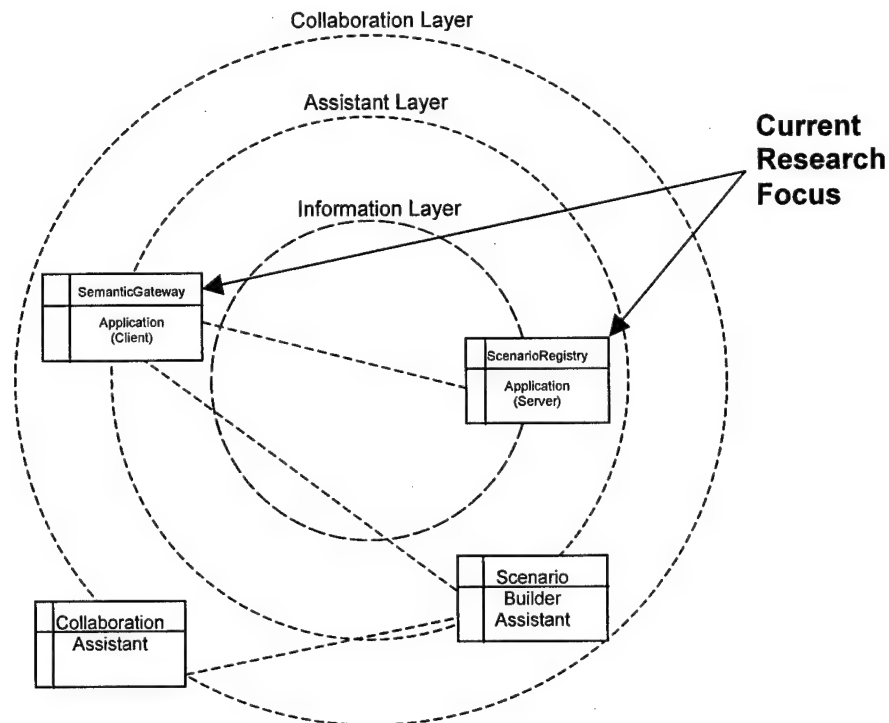


Figure 61: Research Impact on CERTCORT Agent Framework

The tool developed in this research provides agent based scenario component retrieval and contains limited transformation capabilities. A key feature of the system is its extensibility. New simulator scenario source types can be added to the system without modification of existing source code. This extensibility is achieved through an extensive utilization of metadata to provide details on component generation and component transformation. The software developed in this research has furthered the state of the CERTCORT tool.

5.3 State of CERTCORT

With the addition of the Semantic Broker system, the CERTCORT tool is now capable of providing automated facilities for the identification of reusable scenario source components, retrieval of those components, and transformation of components to a selected target format. Figure 62 provides a graphical representation of the state of CERTCORT's functionality.

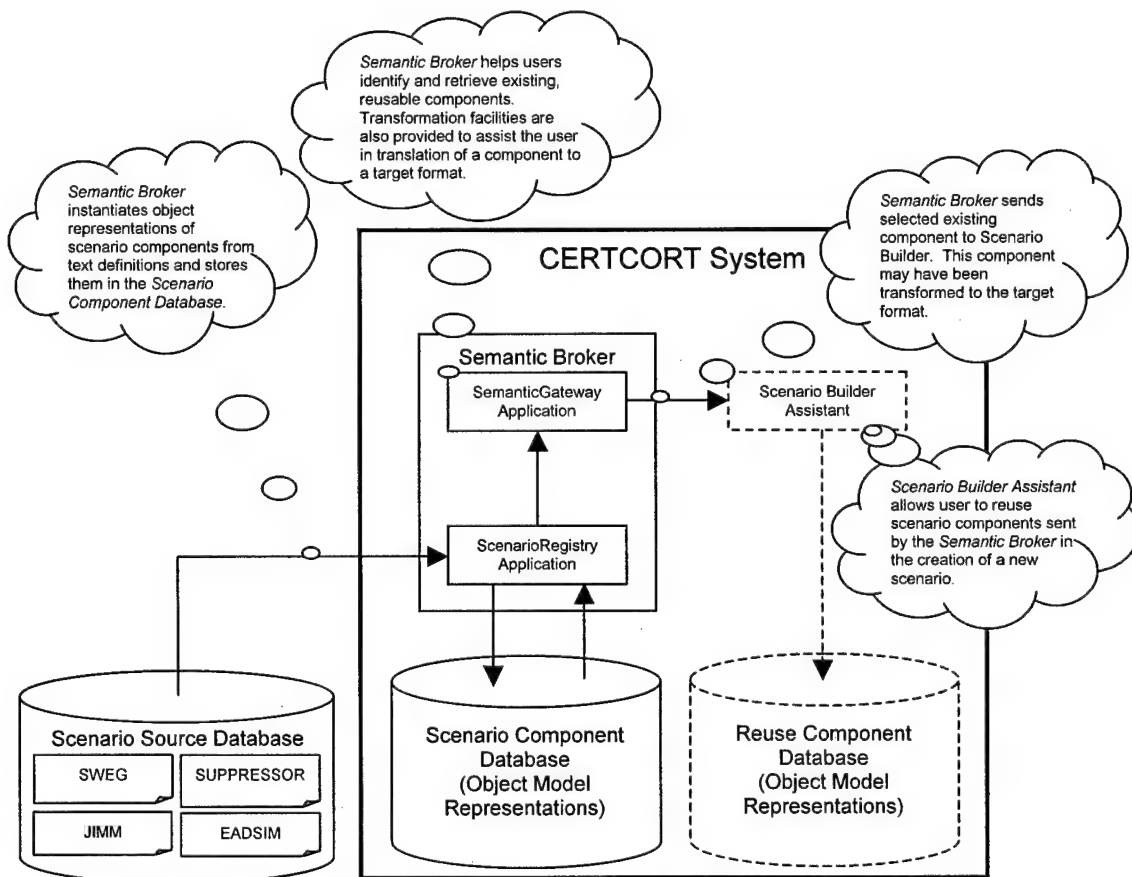


Figure 62: State of CERTCORT Functionality

In the figure, functionality that has not yet been fully developed is denoted by dashed lines around those components. As the figure shows, currently the CERTCORT tool has the following capabilities:

- Instantiating object model scenario component representations from text based scenario files into a common object model.
- Allowing the user to perform signature-based queries to find existing scenario components that are suitable for reuse in a new scenario.
- Assisting the user in the transformation of an existing scenario component to a different scenario format.

The *Scenario Builder Assistant* shown in Figure 62 has yet to be completely developed. A rudimentary version of this assistant was developed in [McD00]. Further development of the *Scenario Builder Assistant*, as well as the *Reuse Component Database*, should be part of future research in the CERTCORT arena.

5.4 Future Research Recommendations

Future research in the CERTCORT area should focus on development of the *Scenario Builder Assistant*, and the inclusion of the Semantic Broker as part of that system. Another area of potential research involves extending the signature analysis concept to operate with entire scenarios as the search signature. Finally, the transformation capabilities developed in this work should be extended to include the Category 4 and 5 transformations discussed in Section 3.3.4, Figure 26.

5.4.1 Developing a Builder Agent

Development of a *Scenario Builder Assistant* is the logical next step in extending the functionality of CERTCORT. This assistant is essentially a scenario design center—a GUI that pulls existing and transformed components together to form a new scenario. A shell of this assistant was developed by [McD00], and this work could form the foundation for the new research.

Since the functions of component retrieval and component transformation are essential to any scenario builder intended to make use of existing scenarios, the

Semantic Broker developed in this work should be included as a subsystem of the *Scenario Builder Assistant*.

5.4.2 Extending the Signature Concept

This research focused on utilizing signature components as search criteria to identify existing scenario source components that fit the requirements of a component needed for a new scenario under construction. A natural extension of this concept is the idea of using entire scenarios as signature components. This would allow users to search for and potentially reuse entire scenarios.

5.4.3 Extending the Semantic Broker's Transformation Capabilities

This work developed the transforms for Category 1, Category 2, Category 3, and Category 6 transformations as defined in Section 3.3.4, Figure 26. Category 4 and Category 5 transforms are beyond the scope of this work. These two transform categories encompass the most difficult aspects of component transformation, since they require the software to interact with the user during the transformation process to determine how to proceed.

5.5 Summary

This work develops a Semantic Broker capable of providing automated relevant component retrieval and component transformation. The foundation of the system is a common object model capable of representing virtually any scenario format, and a technique of utilizing metadata to allow processes to be less format specific.

As a result of this work, the CERTCORT tool under development by AFRL is now capable of maintaining a registry of available source scenarios and their components, identifying relevant scenario components for reuse, retrieving those components from

their distributed locations, and transforming, with certain limitations, those components to a desired target scenario format.

Recommendations for future research in this area include the development of the Scenario Builder Assistant in the CERTCORT Agent Framework's Assistant Layer, extending the signature analysis concept to include entire scenarios, and extending the Semantic Broker's transformation capabilities to include Categories 4 and 5.

Appendix A. Selected Source Code

This appendix contains portions of the source code for the semantic broker system. The code presented here is helpful to understanding the component generation, relevant component retrieval, and the component transformation processes.

A.1. Component Generation

ScenarioRegistryGUI: insertFileTreeNode Method

This method inserts a *RegistryFile* object into the SRDB that represents the scenario source file entered by the user. The file is parsed, and the object models representing its source components are placed in the *Scenario Component Database* of this *ScenarioRegistry* application.

```
public void insertFileTreeNode(SR_DialogInfo o)
{
    RegistryFile newFile = new RegistryFile(o.data1);
    newFile.setParser(o.data2);
    DefaultMutableTreeNode newNode =
        new DefaultMutableTreeNode(newFile);
    DefaultMutableTreeNode currentNode =
        (DefaultMutableTreeNode)tree.
        getLastSelectedPathComponent();
    Object obj = currentNode.getUserObject();
    if (obj instanceof RegistrySource)
    {
        RegistrySource regSource = (RegistrySource)obj;
        RegistryAgent reg =
            (RegistryAgent)((DefaultMutableTreeNode)currentNode.
                getParent()).getUserObject();
        if (!reg.findSourceParser(o.data2))
        {
            displayError("Error",
                "Parser class not found in type node.");
            return;
        }

        // Construct path name
        String srcFile = null, srcParser = null, msfFile;
        msfFile = reg.getMsfFile();
        if (regSource.getLocation().endsWith(File.separator))
            srcFile = regSource.getLocation() +
                newFile.getName();
        else
```

```

        srcFile = regSource.getLocation() + File.separator +
            newFile.getName();

        // Get string representing class name and call
        // getSourceComponents to generate source components
        srcParser = newFile.getParser();
        String[] srcParams = {srcFile, msfFile};
        Vector srcComps = getSourceComponents(srcParser,
            srcParams);

        if (srcComps != null)
        {
            // Load this type's serialized file
            loadSourceComps(reg.getType());
            if (this.sourceComps == null)
                this.sourceComps = new Vector();
            int i;
            // Add new scenario components to serialized database
            for (i = 0; i < srcComps.size(); i++)
                this.sourceComps.add(srcComps.get(i));

            int fileIndex = ((RegistrySource)obj).insertFile(newFile);
            treeModel.insertNodeInto(newNode, currentNode, fileIndex);
            TreeNode[] nodes = treeModel.getPathToRoot(newNode);
            TreePath path = new TreePath(nodes);
            tree.scrollPathToVisible(path);
            saveSourceComps();
        }
    }
}

```

ScenarioRegistryGUI: getSourceComponents Method

This method uses Java's reflection mechanism to instantiate a *Parser* object of the sub-class referenced by the string *p*. After the *Parser* object is instantiated, the method generates scenario components from the file referenced in the *parameters* argument.

```

public Vector getSourceComponents(String p,
                                String[] parameters)
{
    Parser parser = null;
    try
    {
        Class cl = Class.forName(p);
        Constructor[] constructors = cl.getDeclaredConstructors();
        parser = (Parser) constructors[0].newInstance(parameters);
    }
    catch (ClassNotFoundException x)
    {
        System.err.println(x);
        return null;
    }
}

```

```

catch (IllegalAccessException x)
{
    System.err.println(x);
    return null;
}
catch (InvocationTargetException x)
{
    System.err.println(x);
    return null;
}
catch (InstantiationException x)
{
    System.err.println(x);
    return null;
}
// Generate scenario components
Vector syntax = parser.loadMetaSyntax();
Vector comps = parser.generateComponents();
Vector subComps = parser.getSubComponentIndex();
if (subComps != null)
{
    // Add subcomponent index to list
    int i;
    for (i = 0; i < subComps.size(); i++)
        comps.add(subComps.get(i));
}
return comps;
}

```

SemanticGateway: addSignatures Method

This method opens a file chooser dialog and generates signature components from the source file selected by the user.

```

public void addSignatures(String arg)
{
    StringTokenizer t = new StringTokenizer(arg, " ");
    t.nextToken();
    String token = t.nextToken();
    int i;
    RegistryType regObj = null;

    // Find RegistryType object for selected signature type
    for (i = 0; i < this.sourceRegistry.size(); i++)
    {
        String tmp = ((RegistryType)this.sourceRegistry
                     .get(i)).getName();
        if (token.equals(tmp))
            regObj = (RegistryType)this.sourceRegistry.get(i);
    }

    // Get user's signature file via a file chooser dialog
    JFileChooser d = new JFileChooser();
    d.setCurrentDirectory(new File("."));
}

```

```

d.setMultiSelectionEnabled(false);
int result = d.showOpenDialog(this);
if (result == JFileChooser.CANCEL_OPTION)
    return;
File sigFile = d.getSelectedFile();
String sigFilename = sigFile.getPath();

// Extract signature Parser class name and meta syntax
// file name from RegistryType object.
String sigParser = regObj.getSigParser();
String[] sigParams = {sigFilename, regObj.getMsFile()};
Vector sigComps = getSignatures(sigParser, sigParams);
if (sigComps == null)
    return;

// Load existing signatures of selected type and
// consolidate new into existing list.
if (!loadSignatures(regObj.getName()))
    this.signatures = new Vector();
consolidateSignatures(sigComps);
serializeOut(regObj.getName() + ".sig", this.signatures);
}

```

SemanticGateway: getSignatures Method

This method uses Java's reflection mechanism to instantiate a *Parser* object of the sub-class referenced by the string *p*. After the *Parser* object is instantiated, the method generates signature components from the file referenced in the *parameters* argument.

```

public Vector getSignatures(String p,
                           String[] parameters)
{
    Parser parser = null;
    try
    {
        Class cl = Class.forName(p);
        Constructor[] constructors = cl.getDeclaredConstructors();
        parser = (Parser) constructors[0].newInstance(parameters);
    }
    catch (ClassNotFoundException x)
    {
        System.err.println(x);
        return null;
    }
    catch (IllegalAccessException x)
    {
        System.err.println(x);
        return null;
    }
    catch (InvocationTargetException x)
    {
        System.err.println(x);
    }
}

```

```

        return null;
    }
    catch (InstantiationException x)
    {
        System.err.println(x);
        return null;
    }
    Vector syntax = parser.loadMetaSyntax();
    Vector comps = parser.generateComponents();
    Vector subComps = parser.getSubComponentIndex();

    if (subComps == null)
        return null;

    int i;
    for (i = 0; i < subComps.size(); i++)
        comps.add(subComps.get(i));
    return comps;
}

```

A.2. Relevant Component Retrieval

The relevant component retrieval process is initiated by the *SemanticGateway* application, but is carried out almost entirely by the individual *ScenarioRegistry* applications. This section provides the source code for the *retrieveRelevantComponents* method of the *ScenarioRegistryAgent*, the *ComponentAnalyzer* class, and the *analyzeComponents* methods of the *MetaComponent* and *SComponent* classes.

ScenarioRegistryAgent: retrieveRelevantComponents Method

This method calls its parent's (i.e., the *ScenarioRegistryGUI* object) *loadSourceComponents* method with the signature's type as the parameter. That method returns a list of the source components available for the signature's type. This method then creates a *ComponentAnalyzer* object and iteratively passes each source component to it. This method returns a *Vector* containing the source components whose relevance score was higher than the *threshold* parameter.

```

public synchronized Vector retrieveRelevantComponents(
                                MetaComponent signature,
                                double threshold)
{
    this.relevantComps = new Vector(); // Reset
    if (signature == null)

```

```

        return relevantComps;

this.sourceComps = this.parent.
    loadSourceComps(signature.getType());

if (this.sourceComps == null)
    return this.relevantComps;

int j;
Vector srcComps, relComps;

if (this.sourceComps != null)
{
    ComponentAnalyzer analyzer = new ComponentAnalyzer(signature);
    for (j = 0; j < this.sourceComps.size(); j++)
    {
        analyzer.setSourceComponents((MetaComponent)
            this.sourceComps.get(j));
        relComps = analyzer.getRelevantComponents(this.localHost,
            threshold);
        addToRelevant(relComps);
    }
}
return this.relevantComps;
}

```

ScenarioRegistryGUI: loadSourceComps Method

This method checks to see if the source component list needed is currently in memory. If so, a reference to that list is returned. Otherwise, the source components are loaded from their serialized object file.

```

public Vector loadSourceComps(String type)
{
    String filename = type + ".src";

    // Check to see if source component vector is null. If
    // not check to see if source components currently in
    // memory are correct type (i.e. same as signature)
    if (this.sourceComps != null)
    {
        MetaComponent mComp = (MetaComponent)this.sourceComps.get(0);
        if (mComp.getType().equals(type))
            return this.sourceComps;
    }
    try
    {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(filename));

        this.sourceComps = new Vector();
        this.sourceComps = (Vector)in.readObject();
        in.close();
    }
}

```



```

    }
    catch (IOException x)
    {
        this.sourceComps = null;
        System.err.println("Unable to open file: " + filename);
        displayError("File Access Error",
            "Unable to open file: " + filename);
    }
    catch (ClassNotFoundException x)
    {
        this.sourceComps = null;
        System.err.println("Unable to find class.");
        displayError("File Access Error",
            "Unable to find required class definition.");
    }
    return this.sourceComps;
}

```

ComponentAnalyzer Class

This class is utilized to perform relevant component analysis. Its constructor requires one parameter: a signature component. The using process then sets its *sourceComponents* attribute via the *setSourceComponents* method, and calls the *getRelevantComponents* method with the host name and sensitivity threshold as parameters.

```

import java.util.*;
import java.io.*;

public class ComponentAnalyzer
{
    protected MetaComponent signature;
    protected MetaComponent sourceComponents;

    public ComponentAnalyzer(MetaComponent sig)
    {
        this.signature = sig;
        this.sourceComponents = null;
    }

    /*****
    * Method Name: getRelevantComponents
    * Purpose:     Scores the scenario components in the
    *              'sourceComponents' vector based on their
    *              similarity to the signature component.
    * Original:    03 Oct 2000
    * Modified:
    *****/
    public Vector getRelevantComponents(String host,
                                       double threshold)

```

```

{
    if (this.sourceComponents == null)
        return new Vector();
    MetaComponent mComp, sComp;
    Vector rtn = new Vector();
    int i, j;
    mComp = this.sourceComponents;
    if (signature.getType().equalsIgnoreCase(mComp.getType()))
        rtn = mComp.analyzeComponents(signature, host, threshold);
    return rtn;
}

public void setSourceComponents(MetaComponent o)
{
    this.sourceComponents = o;
}
}

```

MetaComponent Class: *analyzeComponents* Method

This method is called by the ComponentAnalyzer class to compare the source object model to the signature object model. It is useful to remember here that in a source object model, the *MetaComponent* root object represents a scenario file. It contains, in its *components* attribute, references to each scenario source component present in the source file from which it was created. This method iteratively calls the *analyzeComponents* method of each of these *SComponent* objects, and, if their score is higher than the *threshold* parameter, creates a *MetaComponent* root for each and places it in its return variable.

```

public Vector analyzeComponents(MetaComponent sig,
                               String host,
                               double threshold)
{
    SComponent signature = sig.getComponent(0);
    Vector rtn = new Vector();
    MetaComponent mComp;
    int i;
    for (i = 0; i < this.components.size(); i++)
    {
        double score = ((SComponent)this.components.get(i)).
            analyzeComponents(signature);
        if (score > threshold)
        {
            mComp = new MetaComponent();
            mComp.setType(sig.getType());
            mComp.setName(((SComponent)this.components.get(i)).
                getType() + " " +

```

```

        ((SComponent)this.components.get(i)).
        getName() + " (" + this.getType() +
        "; Source: System = " + host + ", " +
        this.getSource() + ")";
    mComp.setSource(host + ": " + this.getSource());
    mComp.setComments(((SComponent)this.components.get(i))
        .getComments());
    mComp.setScore(score);
    rtn.add(mComp);
}
}
return rtn;
}

```

SComponent Class: analyzeComponents Method

This method compares the *SComponent* object to the input signature *SComponent* object.

```

public double analyzeComponents(SComponent signature)
{
    double rtn = 0.0;
    // If component types don't match return 0.0
    if (!(this.getType().trim().equalsIgnoreCase(signature.
        getType().trim())))
        return 0.0;
    // Compare source component's characteristics to those
    // of the signature
    double attrScore = analyzeAttributes(signature.
        getCharacteristics());

    // If signature component has no sub-components return the
    // characteristic score alone.
    Vector sigComps = signature.getComponents();
    if (sigComps.size() == 0)
        return attrScore;

    String sigCompType;
    double noMatchingComp = 0.0;
    double tmpMatchCount, hiMatchCount = 0.0;
    boolean found;
    int i, j;
    // Iterate through signature's sub-components and
    // determine whether source component has a sub-component
    // of a matching type. If so, call that sub-component's
    // analyzeComponents method with the signature sub-
    // component as the parameter.
    for (i = 0; i < sigComps.size(); i++)
    {
        hiMatchCount = 0.0;
        found = false;
        sigCompType = ((SComponent)sigComps.get(i)).getType();
        for (j = 0; j < this.components.size(); j++)
        {

```

```

        tmpMatchCount = 0.0;
        if (sigCompType.trim().
            equalsIgnoreCase(((SComponent)this.components.
                               get(j)).getType().trim()))
        {
            tmpMatchCount = (((SComponent)this.components.get(j)).
                               analyzeComponents(((SComponent)sigComps.get(i)));
            found = true;
        }
        if (tmpMatchCount > hiMatchCount)
            hiMatchCount = tmpMatchCount;
    }
    noMatchingComp = noMatchingComp + hiMatchCount;

    // If signature component is mandatory and not found,
    // return zero.
    if (!found &&
        (!((SComponent)sigComps.get(i)).isOptional()))
        return 0.0;
    }
    double compScore = (double)noMatchingComp/sigComps.size();

    // If signature component has no characteristics, return
    // the sub-component score. Otherwise, combine the
    // sub-component and characteristics scores and return
    // their average.
    if (signature.getCharacteristics().size() == 0)
        rtn = compScore;
    else
        rtn = (double)((compScore + attrScore)/2);

    return rtn;
}

```

SComponent Class: analyzeAttributes Method

This method compares the *SComponent* object's *characteristics* attribute to the input contents of the input *Vector* and returns a score that is the ratio of *String* object matches to number of *String* objects in the input *Vector*.

```

public double analyzeAttributes(Vector attributes)
{
    int attributeMatches = 0, i, j;
    String sigAttribute, compAttribute = null;
    boolean found;
    if (attributes.size() == 0)
        return 1.0;
    for (i = 0; i < attributes.size(); i++)
    {
        sigAttribute = (String)attributes.get(i);
        found = false;
        for (j = 0; j < this.characteristics.size() && !found; j++)
        {

```

```

        compAttribute = (String)this.characteristics.get(j);
        if (sigAttribute.trim().
            equalsIgnoreCase(compAttribute.trim()))
        {
            attributeMatches = attributeMatches + 1;
            found = true;
        }
    }
}
return (double)attributeMatches/attributes.size();
}

```

A.3. Component Transformation

This section contains the essential code for the transformation process of the semantic broker. The *transformComponent* method of the *SemanticGateway* class, *TransformEngine* class, *Transform* abstract class, and the *SUPP_SWEG_Xform_1*, *SUPP_SWEG_Xform_2*, *SUPP_SWEG_Xform_3*, and *SUPP_SWEG_Xform_6* class definitions are provided.

SemanticGateway Class: transformComponent Method

This method extracts the required metadata and calls the appropriate methods to transform the selected source component.

```

public SComponent transformComponent(String menuArg)
{
    RegistryTransform regXForm = null;
    DefaultMutableTreeNode selectedNode =
        (DefaultMutableTreeNode)compTree.
            getLastSelectedPathComponent();
    Object obj = selectedNode.getUserObject();
    if (!(obj instanceof MetaComponent))
        return null;

    // Extract source name from menu item selection
    StringTokenizer t = new StringTokenizer(menuArg, " ");
    t.nextToken();
    String sourceName = t.nextToken();

    if (((MetaComponent)obj).getType().
        equalsIgnoreCase(sourceName))
        return null;

    // Extract transform name from menu item selection
    t = new StringTokenizer(menuArg, ":");
    t.nextToken();

```

```

String transformName = t.nextToken().trim();

// Extract source location and get root of component
String sourceLocation = ((MetaComponent)obj).getSource();
SComponent source = ((MetaComponent)obj).getComponent(0);

// Find RegistryTransform object in SWSRDB that contains
// information about this transformation
int i;
for (i = 0; i < this.sourceRegistry.size(); i++)
{
    if (((RegistryType)this.sourceRegistry.get(i)).
        getName().equalsIgnoreCase(sourceName))
        regXForm = ((RegistryType)this.sourceRegistry.get(i)).
            findTransform(transformName);
}
if (regXForm == null)
    return null;

// Create TransformEngine object and transform component
TransformEngine xFormEng = new TransformEngine(regXForm);
SComponent rtn = xFormEng.xFormModel(source);
Vector v = new Vector();
// Create MetaComponent root for transformed object.
MetaComponent root = new MetaComponent();
root.setType(regXForm.getTargetType());
root.setName(regXForm.getTargetType() + ": " +
    source.getName());
root.addComponent(rtn);
Vector com = (source.getComments());
for (i = 0; i < com.size(); i++)
    root.addComment((String)com.get(i));
root.addComment("TRANSFORMATION INFORMATION:\n" +
    "This scenario component was transformed by the " +
    "CERTCORT SemanticGateway agent. The " +
    "original component and location were: \n" +
    "SCENARIO TYPE: " + ((MetaComponent)obj).getType()
    + "\n" + "COMPONENT: " + source.getType() + "\n" +
    "SYSTEM AND PATH: " +
    ((MetaComponent)obj).getSource());
root.setSignature(true);
v.add(root);

// Create ComponentViewer and display transformed object.
Frame f = new ComponentViewer(v, this, transformName +
    " Transformation");

f.show();
return rtn;
}

```

TransformEngine Class

This class provides the necessary data structures and methods to facilitate component transformations.

```

/*****
* Source file: TransformEngine.java
* Purpose:
*
* History:
*   Original: 19 Oct 2000, Breighner
*   Modified:
*****/
import java.util.*;
import java.io.*;
import java.lang.reflect.*;

public class TransformEngine
{
    protected String sourceType;
    protected String targetType;
    protected SComponent sourceComp;
    protected Vector xFormMD;
    protected Vector xForms;
    protected RegistryTransform regXform;

    public TransformEngine(RegistryTransform o)
    {
        this.regXform = o;
        this.sourceType = o.getSourceType();
        this.targetType = o.getTargetType();
        this.xForms = getTransforms();
        MDPParser parser = new MDPParser(o.getXFormMDFile());
        this.xFormMD = parser.loadXFormMetadata();
    }

/*****
* Method Name: xFormModel
* Purpose:      Transforms the scenario component model whose
*               root is input SComponent object 'o.'
* Original:     19 Oct 2000
* Modified:
*****/
    public SComponent xFormModel(SComponent o)
    {
        this.sourceComp = o;
        SComponent rtn = null;

        // Transform input parameter 'o'
        rtn = xFormComp(o);

        // Transform the sub-components of input parameter 'o'
        // and add them to the transformed component.
        int i;
        Vector comps = o.getComponents();
        for (i = 0; i < comps.size(); i++)
            rtn.addComponent(xFormModel((SComponent)comps.get(i)));
        return rtn;
    }
}

```

```

/*****
* Method Name: xFormComp
* Purpose:      Transform input SComponent 'o.'
* Original:     19 Oct 2000
* Modified:
*****/
public SComponent xFormComp(SComponent o)
{
    SComponent rtn;
    // Find transform metadata object. If not found
    // abort component transformation.
    TransformMD xformMD = findXFormMD(o.getType());
    if (xformMD == null)
        return null;

    // Retrieve Transform sub-class object based on category
    // data contained in transformation metadata object. If
    // transform is null abort.
    Transform xform = (Transform)this.xForms.
        get(xformMD.getCategory() - 1);
    if (xform == null)
        return null;

    // Set transform metadata, then call Transform object's
    // transform method with source component as parameter.
    xform.setXFormMD(xformMD);
    rtn = xform.transform(o);
    return rtn;
}

/*****
* Method Name: getTransforms
* Purpose:      Instantiates the transform classes from the
*               String objects stored in the RegistryTransform
*               object 'regXform.'
* Original:     19 Oct 2000
* Modified:
*****/
public Vector getTransforms()
{
    Transform transform = null;
    Vector rtn = new Vector();
    int i;
    Vector xFormNames = this.regXform.getXFormClasses();
    for (i = 0; i < xFormNames.size(); i++)
    {
        try
        {
            Class cl = Class.forName((String)xFormNames.get(i));
            Constructor[] constructors = cl.getDeclaredConstructors();
            String[] parameters = {regXform.getCommentDelimiter()};
            transform = (Transform)constructors[0].newInstance(parameters);
        }
        catch (ClassNotFoundException x)
        {

```



```

        System.err.println(x);
        return null;
    }
    catch (IllegalAccessException x)
    {
        System.err.println(x);
        return null;
    }
    catch (InvocationTargetException x)
    {
        System.err.println(x);
        return null;
    }
    catch (InstantiationException x)
    {
        System.err.println(x);
        return null;
    }
    rtn.add(transform);
}
return rtn;
}

/*****
* Accessors and Mutators
* Original: 19 Oct 2000, Breighner
* Modified:
*****/
public void setSourceType(String s)
{
    this.sourceType = s;
}

public void setTargetType(String s)
{
    this.targetType = s;
}

public void setXFormMD(Vector v)
{
    this.xFormMD = v;
}

public String getSourceType()
{
    return this.sourceType;
}

public String getTargetType()
{
    return this.targetType;
}

public Vector getXFormMD()
{

```

```

        return this.xFormMD;
    }

    public TransformMD findXFormMD(String s)
    {
        int i;
        String b;
        for (i = 0; i < this.xFormMD.size(); i++)
        {
            b = ((TransformMD)this.xFormMD.get(i)).getSourceType();
            if (b.equals(s))
                return (TransformMD)this.xFormMD.get(i);
        }
        return null;
    }
}

```

Transform Class

This abstract class provides the foundation for all transformation sub-classes. All transform classes must extend this class and implement their version of the *transform* method.

```

/*****
* Source file: Transform.java
* Purpose:      This abstract class is the super class of all
*               transforms in the SemanticGateway. All transform
*               classes must extend this class.
*
* History:
*   Original:   18 Oct 2000, Breighner
*   Modified:
*****/
import java.util.*;
import java.io.*;

public abstract class Transform
{
    protected String category;
    protected TransformMD xFormMD;
    protected String commentDelimiter;

    public Transform(String s, String delimiter)
    {
        this.category = s;
        this.xFormMD = null;
        this.commentDelimiter = delimiter;
    }

/*****
* Method Name: transform
* Purpose:      Abstract method. Must be implemented in
*               sub-class.
* Original:     18 Oct 2000, Breighner

```

```

* Modified:
*****/
public abstract SComponent transform(SComponent o);

/*****
* Accessors and Mutators
* Original: 18 Oct 2000, Breighner
* Modified:
*****/
public void setCategory(String s)
{
    this.category = s;
}

public void setXFormMD(TransformMD o)
{
    this.xFormMD = o;
}

public void setCommentDelimiter(String s)
{
    this.commentDelimiter = s;
}

public String getCategory()
{
    return this.category;
}

public TransformMD getXFormMD()
{
    return this.xFormMD;
}

public String getCommentDelimiter()
{
    return this.commentDelimiter;
}

public String toString()
{
    return new String(this.category);
}
}

```

The following four classes extend the Transform class and map to the Category 1, 2, 3, and 6 transforms discussed in Chapter 3. Category 4 and 5 transformations are beyond the scope of this work. Components that fall in those categories are treated as Category 6 transformations.

SUPP_SWEG_Xform_1 Class

```

/*****
* Source file: SUPP_SWEG_Xform_1.java
* Purpose:      Transform for Category 1 SUPPRESSOR-to-SWEG
*               translations.
*
* History:
*   Original:   24 Oct 2000, Breighner
*   Modified:
*****/
import java.util.*;
import java.io.*;

public class SUPP_SWEG_Xform_1 extends Transform
{
    public SUPP_SWEG_Xform_1(String commentD)
    {
        super("CATEGORY 1", commentD);
    }

    /*****
    * Method Name: transform
    * Purpose:      Transforms input parameter 'o' from SUPPRESSOR
    *               format to SWEG format
    * Original:     18 Oct 2000, Breighner
    * Modified:
    *****/
    public SComponent transform(SComponent o)
    {
        SComponent rtn = new SComponent();
        rtn.setType(o.getType());
        rtn.setName(o.getName());
        rtn.setCharacteristics(o.getCharacteristics());
        return rtn;
    }
}

```

SUPP_SWEG_Xform_2 Class

```

/*****
* Source file: SUPP_SWEG_Xform_2.java
* Purpose:      Transform for Category 2 SUPPRESSOR-to-SWEG
*               translations.
*
* History:
*   Original:   24 Oct 2000, Breighner
*   Modified:
*****/
import java.util.*;
import java.io.*;

public class SUPP_SWEG_Xform_2 extends Transform
{

```

```

public SUPP_SWEG_Xform_2(String commentD)
{
    super("CATEGORY 2", commentD);
}

/*****
* Method Name: transform
* Purpose:      Transforms input parameter 'o' from SUPPRESSOR
*               format to SWEG format
* Original:     18 Oct 2000, Breighner
* Modified:
*****/
public SComponent transform(SComponent o)
{
    SComponent rtn = new SComponent();
    rtn.setType(o.getType());
    rtn.setName(o.getName());
    rtn.setCharacteristics(o.getCharacteristics());
    int i;
    String compChar = null, charXFormName = null;
    TransformMD charXForm = null;
    Vector charXForms = this.xFormMD.getSubComponentXForms();
    for (i = 0; i < charXForms.size(); i++)
    {
        charXForm = (TransformMD)charXForms.get(i);
        charXFormName = charXForm.getSourceType();
        if (charXFormName.equalsIgnoreCase("NOT-IN-SOURCE"))
        {
            rtn.addComment(this.commentDelimiter +
                           " ITEM NOT AVAILABLE IN SOURCE " +
                           charXForm.getTargetType());
        }
    }
    return rtn;
}
}

```

SUPP_SWEG_Xform_3 Class

```

/*****
* Source file: SUPP_SWEG_Xform_3.java
* Purpose:      Transform for Category 3 SUPPRESSOR-to-SWEG
*               translations.
*
* History:
*   Original:   24 Oct 2000, Breighner
*   Modified:
*****/
import java.util.*;
import java.io.*;

public class SUPP_SWEG_Xform_3 extends Transform
{
    public SUPP_SWEG_Xform_3(String commentD)

```

```

{
    super("CATEGORY 3", commentD);
}

/*****
* Method Name: transform
* Purpose:      Transforms input parameter 'o' from SUPPRESSOR
*               format to SWEG format
* Original:     18 Oct 2000, Breighner
* Modified:
*****/
public SComponent transform(SComponent o)
{
    SComponent rtn = new SComponent();
    rtn.setType(this.xFormMD.getTargetType());
    rtn.setName(o.getName());
    int i, j;
    StringTokenizer t;
    String compChar = null, charXFormName = null,
        tmpChar = null;
    TransformMD charXForm = null;
    Vector compChars = o.getCharacteristics();
    Vector charXForms = this.xFormMD.getSubComponentXForms();
    for (j = 0; j < compChars.size(); j++)
    {
        t = new StringTokenizer((String)compChars.get(j), " ");
        tmpChar = "";
        TransformMD subMD;
        while (t.hasMoreTokens())
        {
            String token = t.nextToken();
            if ((subMD =
                xFormMD.findSubComponentXForm(token)) == null)
                tmpChar = tmpChar + " " + token;
            else
            {
                if (!tmpChar.equals(""))
                {
                    rtn.addCharacteristic(tmpChar.trim());
                    tmpChar = "";
                }
                tmpChar = subMD.getTargetType();
                if (tmpChar.equalsIgnoreCase("NOT-IN-TARGET"))
                    tmpChar = this.commentDelimiter +
                        " ITEM NOT AVAILABLE IN TARGET FORMAT: " +
                        subMD.getSourceType();
            }
        }
        rtn.addCharacteristic(tmpChar.trim());
    }
    for (i = 0; i < charXForms.size(); i++)
    {
        charXForm = (TransformMD)charXForms.get(i);
        charXFormName = charXForm.getSourceType();
        if (charXFormName.equalsIgnoreCase("NOT-IN-SOURCE"))
    }

```

```

        {
            rtn.addCharacteristic(this.commentDelimiter +
                                " ITEM NOT AVAILABLE IN SOURCE FORMAT: " +
                                charXForm.getTargetType());
        }
    }
    return rtn;
}
}

```

SUPP_SWEG_Xform_6 Class

```

/*****
* Source file: SUPP_SWEG_Xform_4.java
* Purpose:      Transform for Category 4 SUPPRESSOR-to-SWEG
*               translations.
*
* History:
*   Original:   24 Oct 2000, Breighner
*   Modified:
*****/
import java.util.*;
import java.io.*;
public class SUPP_SWEG_Xform_4 extends Transform
{
    public SUPP_SWEG_Xform_4(String commentD)
    {
        super("CATEGORY 4", commentD);
    }

    /*****
    * Method Name: transform
    * Purpose:      Transforms input parameter 'o' from SUPPRESSOR
    *               format to SWEG format
    * Original:     18 Oct 2000, Breighner
    * Modified:
    *****/
    public SComponent transform(SComponent o)
    {
        SComponent rtn = new SComponent();
        rtn.setType(this.commentDelimiter +
                    "NOT AVAILABLE IN TARGET FORMAT: " + o.getType());
        rtn.setName(o.getName());
        rtn.setCharacteristics(o.getCharacteristics());
        rtn.setComponents(o.getComponents());
        return rtn;
    }
}

```

Appendix B. Metadata

This appendix provides some insight into the actual content of the metadata files utilized by the *Parser* and *Transform* objects.

SUPPRESSOR.MSF

This file contains all the scenario component syntax definitions for the SUPPRESSOR source type. This file is accessed by the *Parser* object to build a list of MetaSyntaxUnit objects. The *Parser* object then references this list during parsing to determine how to interpret a given component.

```
PLAYER-STRUCTURE attribute component END PLAYER-STRUCTURE
TACTIC component END TACTIC
CAPABILITY component END CAPABILITY
LINKAGES attribute NULL
SUSCEPTIBILITY component END SUSCEPTIBILITY
ASG-CMD-CHAIN attribute NULL
EVALUATION-RATES attribute END EVALUATION-RATES
INTELL-REPORT-FREQ attribute END INTELL-REPORT-FREQ
MAX-MSG-ATTEMPTS attribute NULL
MAX-SNR-PERCEPTIONS attribute NULL
MOVE-TO-ENG attribute NULL
MSG-RPT-GUIDE attribute END MSG-RPT-GUIDE
SALVO-FIRING attribute END SALVO-FIRING
SNR-RPT-GUIDE attribute END SNR-RPT-GUIDE
ZONE-CHARACTERISTICS attribute END ZONE-CHARACTERISTICS
RESOURCE-ALLOCATION component END RESOURCE-ALLOCATION
LETHAL-ENGAGE-QUEUE-ADD attribute END LETHAL-ENGAGE-QUEUE-ADD
LETHAL-ENGAGE-QUEUE-DROP attribute END LETHAL-ENGAGE-QUEUE-DROP
LETHAL-ENGAGE-START attribute END LETHAL-ENGAGE-START
LETHAL-ENGAGE-STOP attribute END LETHAL-ENGAGE-STOP
LETHAL-ENGAGE-FIRING-START attribute END LETHAL-ENGAGE-FIRING-START
LETHAL-ENGAGE-FIRING-STOP attribute END LETHAL-ENGAGE-FIRING-STOP
SUSCEPTIBILITY attribute END SUSCEPTIBILITY
IR-RAD-TABLE attribute END IR-RAD-TABLE
OPT-CS attribute END OPT-CS
INHERENT-CONTRAST attribute NULL
TGT-REFLECTIVITY attribute END TGT-REFLECTIVITY
RCS-TABLE attribute END RCS-TABLE
SNR-ELE-INTERACTIONS attribute END SNR-ELE-INTERACTIONS
CAPABILITY component END CAPABILITY
NUM-SIMULTANEOUS-ROUND attribute NULL
PLATFORM-VEL-ATTEN attribute NULL
RESOURCE-DISAGGREGATION attribute END RESOURCE-DISAGGREGATION
WPN-CHARACTERISTICS attribute END WPN-CHARACTERISTICS
WPN-PK attribute END WPN-PK
WPN-SPD-CAPABILITY attribute END WPN-SPD-CAPABILITY
```


WPN-TIME-DELAYS attribute END WPN-TIME-DELAYS
 WPN-TIME-DELAY-TABLE attribute END WPN-TIME-DELAY-TABLE
 HITS-TO-ESTABLISH-TRACK attribute NULL
 ONE-M2-DETECT-RNG attribute NULL
 PEAK-GAIN attribute NULL
 EFFECTIVE-EARTH-RADIUS attribute NULL
 VERTICAL-OFFSET attribute NULL
 RCVR-BANDWIDTH attribute NULL
 SENSING-MODE-RATES attribute END SENSING-MODE-RATES
 DETECTION-SENSITIVITIES attribute END DETECTION-SENSITIVITIES
 MTI-ATTENUATION attribute END MTI-ATTENUATION
 ANTENNA-PATTERN attribute END ANTENNA-PATTERN
 SNR-CHARACTERISTICS attribute END SNR-CHARACTERISTICS
 QUALITY-OF-DATA attribute END QUALITY-OF-DATA
 RNG-ALT-CAPABILITY attribute END RNG-ALT-CAPABILITY
 INTERNAL-LOSS attribute NULL
 PEAK-POWER-OUTPUT attribute NULL
 PULSE-REPETITION-FREQ attribute NULL
 XMIT-FREQ attribute NULL
 MAX-PARALLEL-TRACKS attribute NULL
 EFF-BURST-CM-PROB attribute NULL
 SNR-TRACKING-PROBABILITIES attribute END SNR-TRACKING-PROBABILITIES
 SNR-TIME-DELAYS attribute END SNR-TIME-DELAYS
 SNR-DOPPLER-LIMITS attribute END SNR-DOPPLER-LIMITS
 IMPLICIT-CM-INTERACT attribute END IMPLICIT-CM-INTERACT
 TIME-BEFORE-DROP attribute NULL
 TIME-TO-THINK attribute END TIME-TO-THINK
 ACCELERATION-MODE attribute NULL
 REVECTOR-DIST-THRESH attribute END REVECTOR-DIST-THRESH
 ATK-PRIORITIES attribute END ATK-PRIORITIES
 MOVE-PLANS component END MOVE-PLANS
 PLAN attribute END-PLAN
 PLAN-PROFILE attribute END PLAN-PROFILE
 SNR-ANGULAR-LIMITS attribute END SNR-ANGULAR-LIMITS
 MAX-ACCELERATION attribute NULL
 MIN-TURN-RADIUS attribute NULL
 MOVER-ALTITUDE-LIMITS attribute END MOVER-ALTITUDE-LIMITS
 MOVER-CLIMB/DIVE-LIMITS attribute END MOVER-CLIMB/DIVE-LIMITS
 MOVER-SPEED-LIMITS attribute END MOVER-SPEED-LIMITS
 FUEL-USAGE attribute END FUEL-USAGE
 COMM-LOSS-DECENT-TIME attribute NULL
 RELOAD-CHARACTERISTICS attribute END RELOAD-CHARACTERISTICS
 CENTRALIZATION-THRESHOLDS attribute END CENTRALIZATION-THRESHOLDS
 NOMINAL-SUB-REACT-TIME attribute NULL
 LETHAL-ASSIGNMENT-QUEUE-ADD attribute END LETHAL-ASSIGNMENT-QUEUE-ADD
 LETHAL-ASSIGNMENT-QUEUE-DROP attribute END LETHAL-ASSIGNMENT-QUEUE-DROP
 LETHAL-ASSIGNMENT-START attribute END LETHAL-ASSIGNMENT-START
 LETHAL-ASSIGNMENT-STOP attribute END LETHAL-ASSIGNMENT-STOP
 MOVE-OPTIONS attribute END MOVE-OPTIONS
 THINKER componentRef NULL
 SNR-RCVR componentRef NULL
 SNR-XMTR componentRef NULL
 WEAPON componentRef componentRef NULL
 ORDNANCE attribute NULL
 FUTURE-PLAYER attribute NULL

```

MOVER componentRef componentRef NULL
COMM-RCVR componentRef NULL
COMM-XMTR componentRef NULL
DISRUPTOR componentRef NULL
LAUNCH-CMD-CHAIN attribute END LAUNCH-CMD-CHAIN
PLAN-PATTERNS attribute END PLAN-PATTERNS
BACKGROUND-RADIANCE attribute END BACKGROUND-RADIANCE
PATH-RADIANCE attribute END PATH-RADIANCE
SEARCH-GLIMPSE-PROB attribute END SEARCH-GLIMPSE-PROB
REACQ-GLIMPSE-PROB attribute END REACQ-GLIMPSE-PROB
TRACK-GLIMPSE-PROB attribute END TRACK-GLIMPSE-PROB
PIXEL-FIELD-OF-VIEW attribute NULL
SOLAR-IRRADIANCE attribute END SOLAR-IRRADIANCE
ASG-EVAL-RATE attribute NULL
ASG-TGT-UPDATE-RATE attribute NULL
LAUNCH-EVAL-RATE attribute NULL
LAUNCH-START attribute END LAUNCH-START
GUNS-FREE attribute END GUNS-FREE
GUNS-TIGHT attribute END GUNS-TIGHT
JAMMER-QUEUE-ADD attribute END JAMMER-QUEUE-ADD
JAMMER-QUEUE-DROP attribute END JAMMER-QUEUE-DROP
JAMMER-SPOT-ADD attribute END JAMMER-SPOT-ADD
JAMMER-SPOT-DROP attribute END JAMMER-SPOT-DROP
MAX-NO-SPOTS attribute NULL
MAX-POWER-OUT attribute NULL
MAX-RNG attribute NULL
DISRUPTOR-CHARACTERISTICS attribute END DISRUPTOR-CHARACTERISTICS
DISRUPTOR-FREQ-LIMITS attribute END DISRUPTOR-FREQ-LIMITS
ANTGR-PATTERN attribute END ANTGR-PATTERN
EMCON/TURN-ON attribute END EMCON/TURN-ON
EMCON/TURN-OFF attribute END EMCON/TURN-OFF
LOOK-AHEAD-DISTANCE attribute NULL
THREAT-VOLUME attribute END THREAT-VOLUME
RCVR-NOISE attribute NULL
RECOGNITION-THRESH attribute NULL
POLARIZATION-EFFECTS attribute END POLARIZATION-EFFECTS
COMM-JMR-INTERACTIONS attribute END COMM-JMR-INTERACTIONS
XMTR-BANDWIDTH attribute NULL
XMTR-POWER attribute NULL
SNR-JMR-INTERACTIONS attributes END SNR-JMR-INTERACTIONS
TRANSMISSION-LOSS attribute END TRANSMISSION-LOSS
INTERCEPT-INTERACT attribute END INTERCEPT-INTERACT
END METASYNTAX

```

SUPP_SWEG.XFM

This file places each SUPPRESSOR scenario component into a transformation category, and provides details pertaining to its translation into the target category (i.e., SWEG).

```

CATEGORY 1
  PLAYER-STRUCTURE

```

LINKAGES
FUEL
ELEMENT
THINKER
CAPABILITY
COMM-XMTR
COMM-RCVR
SNR-RCVR
SNR-XMTR
MOVER
WEAPON
ORDNANCE
DISRUPTOR
FUTURE-PLAYER
TIME-BEFORE-DROP
MAX-SNR-PERCEPTIONS
RESOURCE-ALLOCATION
TACTIC
SUSCEPTIBILITY
SNR-ELE-INTERACTIONS
NUM-SIMULTANEOUS-ROUND
PLATFORM-VEL-ATTEN
RESOURCE-DISAGGREGATION
WPN-CHARACTERISTICS
WPN-PK
WPN-SPD-CAPABILITY
WPN-TIME-DELAYS
HITS-TO-ESTABLISH-TRACK
ONE-M2-DETECT-RNG
EFFECTIVE-EARTH-RADIUS
VERTICAL-OFFSET
MAX-PARALLEL-TRACKS
RCVR-BANDWIDTH
ANTENNA-PATTERN
SNR-TRACKING-PROBABILITIES
SNR-DOPPLER-LIMITS
SNR-ANGULAR-LIMITS
SNR-TIME-DELAYS
XMTR-BANDWIDTH
XMTR-POWER
RCVR-BANDWIDTH
DETECTION-SENSITIVITIES
EFF-BURST-CM-PROB
MTI-ATTENUATION
SNR-CHARACTERISTICS
RNG-ALT-CAPABILITY
SNR-TIME-DELAYS
SNR-DOPPLER-LIMITS
INTERNAL-LOSS
PEAK-POWER-OUTPUT
PULSE-REPETITION-FREQ
IMPLICIT-CM-INTERACT
TIME-BEFORE-DROP
SNR-ANGULAR-LIMITS
MAX-ACCELERATION

MIN-TURN-RADIUS
 MOVER-ALTITUDE-LIMITS
 MOVER-CLIMB/DIVE-LIMITS
 MOVER-SPEED-LIMITS
 FUEL-USAGE
 COMM-LOSS-DECENT-TIME
 CENTRALIZATION-THRESHOLDS
 RELOAD-CHARACTERISTICS
 NOMINAL-SUB-REACT-TIME
 MOVE-OPTIONS
 LAUNCH-CMD-CHAIN
 MAX-NO-SPOTS
 MAX-POWER-OUT
 DISRUPTOR-CHARACTERISTICS
 MAX-RNG
 INTERCEPT-INTERACT
 LOOK-AHEAD-DISTANCE
 RCVR-NOISE
 RECOGNITION-THRESH
 POLARIZATION-EFFECTS
 COMM-JMR-INTERACTIONS
 XMTR-BANDWIDTH
 XMTR-POWER
 SNR-JMR-INTERACTIONS
 TRANSMISSION-LOSS
 LOOK-AHEAD-DISTANCE
 THREAT-VOLUME
 END CATEGORY
 CATEGORY 2
 QUALITY-OF-DATA
 AGE-OF-PLATFORM
 PLAYER-NAME
 PLATFORM-ID
 END QUALITY-OF-DATA
 END CATEGORY
 CATEGORY 3
 LOCATION PLATFORM
 END LOCATION
 TIME-TO-THINK TIME-TO-THINK
 EVAL-LETHAL-ENGAGE EVAL-LETHAL-ENGAGE
 EVAL-FIRING EVAL-FIRING
 EVAL-ENGAGE-THREAT EVAL-ENGAGE-THREAT
 RECOG-MSG RECOG-MSG
 RECOG-SNR-EVENT RECOG-SNR-EVENT
 RECOG-PHYS-EVENT RECOG-PHYS-EVENT
 REVIEW-INFORMATION REVIEW-INFORMATION
 EVAL-ASSIGN-THREAT EVAL-ASSIGN-THREAT
 ASSIMILATE-INTELL NOT-IN-TARGET
 CONSIDER-ASG/CANCEL EVAL-ASG/CANCEL
 EVAL-EMCON-CHANGE EVAL-EMCON-CHANGE
 EVAL-JMR-QUEUE EVAL-JMR-QUEUE
 EVAL-JMR-SPOTS EVAL-JMR-SPOTS
 NOT-IN-SOURCE DIGEST-ATTACK
 NOT-IN-SOURCE DIGEST-DEATH
 NOT-IN-SOURCE DIGEST-INTELL

NOT-IN-SOURCE	DIGEST-LOSS-COMM
NOT-IN-SOURCE	DIGEST-MATERIEL-STAT
NOT-IN-SOURCE	DIGEST-MSG
NOT-IN-SOURCE	EVAL-COMM-METHOD
NOT-IN-SOURCE	EVAL-INTELL-SEND
NOT-IN-SOURCE	EVAL-MNVR
NOT-IN-SOURCE	EVAL-MNVR-QUEUE
NOT-IN-SOURCE	EVAL-REQUEST
NOT-IN-SOURCE	CONSIDER-PLAN
END TIME-TO-THINK	
EVALUATION-RATES	EVALUATION-RATES
ASG-EVAL-RATE	ASG-EVAL-RATE
EMCON-EVAL-RATE	EMCON-EVAL-RATE
ENG-EVAL-RATE	ENG-EVAL-RATE
JAM-EVAL-RATE	JAM-EVAL-RATE
NOT-IN-SOURCE	MOVE-EVAL-RATE
NOT-IN-SOURCE	REQUEST-EVAL-RATE
END EVALUATION-RATES	
IR-RAD-TABLE	REFLECTIVE-EM-SIG-TABLE
DIMENSION	DIMENSION
IR-RAD	SIGNATURE
END IR-RAD-TABLE	
OPT-CS	REFLECTIVE-EM-SIG-TABLE
DIMENSION	DIMENSION
OCS	SIGNATURE
IR-RAD	SIGNATURE
RCS	RCS
END OPT-CS	
TGT-REFLECTIVITY	REFLECTIVE-EM-SIG-TABLE
DIMENSION	DIMENSION
REFLECTANCE	SIGNATURE
END TGT-REFLECTIVITY	
RCS-TABLE	REFLECTIVE-EM-SIG-TABLE
DIMENSION	DIMENSION
RCS	SIGNATURE
END RCS-TABLE	
SENSING-MODE-RATES	SENSING-MODE-RATES
TRACK-SENSING-RATE	TRACK-SENSING-RATE
ACQ-SENSING-RATE	ACQ-SENSING-RATE
SEARCH-SENSING-RATE	SEARCH-SENSING-RATE
GUIDANCE-SENSING-RATE	GUIDANCE-SENSING-RATE
FIRING-SENSING-RATE	NOT-IN-TARGET
END SENSING-MODE-RATES	
END CATEGORY	
CATEGORY 6	
INTELL-REPORT-FREQ	
ASG-CMD-CHAIN	
MAX-MSG-ATTEMPTS	
MOVE-TO-ENG	
MSG-RPT-GUIDE	
SALVO-FIRING	
SNR-RPT-GUIDE	
ZONE-CHARACTERISTICS	
INHERENT-CONTRAST	
WPN-TIME-DELAY-TABLE	

PEAK-GAIN
XMIT-FREQ
ACCELERATION-MODE
REVECTOR-DIST-THRESH
ATK-PRIORITIES
MOVE-PLANS
PLAN-PATTERNS
BACKGROUND-RADIANCE
PATH-RADIANCE
SEARCH-GLIMPSE-PROB
REACQ-GLIMPSE-PROB
TRACK-GLIMPSE-PROB
PIXEL-FIELD-OF-VIEW
SOLAR-IRRADIANCE
ANTGR-PATTERN
END CATEGORY
END METADATA

BIBLIOGRAPHY

- [Ash00] Ashby, M. *Tool-Based Integration and Code Generation of Object Models*. MS Thesis, Air Force Institute of Technology (AU), Wright Patterson AFB, OH, AFIT/GCS/ENG/00M-02, Mar 2000.
- [Bla98] Blaha, M., Premerlani, W. Object-Oriented Modeling and Design for Database Applications. Upper Saddle River, NJ, Prentice Hall, 1998.
- [Cam91] Cammarata, S., Shane, D., Ram, P. IID: An Intelligent Information Dictionary for Managing Semantic Metadata. Santa Monica, CA: RAND, 1991.
- [Cam95] Cammarata, S., Kameny, I., Lender, J., Replogle, C. The RAND Metadata Management System (RMMS): A Metadata Storage Facility to Support Data Interoperability, Reuse, and Sharing. Santa Monica, CA: RAND, 1995.
- [Col99] Colonese, E. *Developing a Methodology for Integrating Simulation System Scenario Schemas*. MS Thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, AFIT/GCS/ENG/99J-03, Jun 1999.
- [DeL99] DeLoach, S. and M. Wood *Multiagent Systems Engineering: The Analysis Phase*. Unpublished document. Air Force Institute of Technology (AU), Wright-Patterson AFB, OH. Jun 2000.
- [DWR99] DeLoach, S., M. Wood, and D. Robinson *Designing Multiagent Systems Using Multiagent Systems Engineering Methodology*. Unpublished document. Air Force Institute of Technology (AU), Wright-Patterson AFB, OH.
- [Far98] Farley, J. Java Distributed Computing. Sebastopol, CA O' Reilly & Associates, Inc. 1998.
- [Gam95] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Longman, Inc. 1995.
- [Gle90] Gleason, G. *Semantic Query Optimization in an Object-Oriented Semantic Association Model (OSAM)*. MS Thesis, University of Florida, AFIT/CI/CIA-89-182, 1990.
- [Hod98] Hodge, G. *CENDI Metadata Initiatives: Beyond the Bibliographic Record*. Information International Associates, Inc. Oak Ridge, TN: Apr 1998

- [Kro99] Kroenke, D. Database Processing: Fundamentals, Design, & Implementation. Seventh Edition, Upper Saddle River, NJ: Prentice Hall, 1999.
- [Lin92] Lindsey, D. *A Framework for Classifying and Resolving Semantic Conflicts Using the Enhanced Entity-Relationship Model*. MS Thesis, Naval Postgraduate School, Monterey, CA, Sep 1992.
- [LSA98] LSA Inc. "JIMM Design Document for the Database Conversion Requirement Spiral." JSF Program Office, Aug 1998.
- [McD00] McDonald, J. *Agent Based Framework for Collaborative Engineering Model Development*. MS Thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, AFIT/GCS/ENG/00M-16, Mar 2000.
- [Mul97] Muller, P. Instant UML. Birmingham, UK: Wrox Press Ltd, 1997.
- [SAIC97] Science Applications International Corporation. "Suppressor Release 5.4 User's Guide, Volume I, II, and III." Jun 1997.
- [Ste99] Steenbarger, M. *Climbing a JTree for the First Time*. Java Developer's Journal. Volume 4, Issue 4, 1999.
- [Str99] Stratton, P. *A Metrics-Based Analysis of Interface Usability Improvements by Applying Intelligent Agents*. MS Thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, AFIT/GCS/ENG/99M-18, Mar 1999.
- [Sub98] Subrahmanian, V. Asynchronous, Distributed, Scalable, Algorithms for Intelligent Reasoning with Geographically Dispersed, Hybrid Knowledge Bases. Contract F30602-93-C-0241. University of Maryland, 1998.
- [Web99] Weber, R. *Extracting a Common Object Model for DOD Simulation Systems*. MS Thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, AFIT/GCS/ENG/99M-20, Mar 1999.
- [Wei99] Weiss, G. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. Cambridge, MA: The MIT Press, 1999.
- [WoD00] Wood M. and S. Deloach *An Overview of the Multiagent Systems Engineering Methodology*. Proceedings of The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000). Limerick, Ireland, June 10, 2000.
- [GrT96] Grassman, W. and J. Tremblay Logic and Discrete Mathematics A Computer Science Perspective. Upper Saddle River, NJ. Prentice Hall, 1996.

VITA

Captain Lawrence A. Breighner enlisted in U.S. Air Force in 1987 and was trained as an F-16 avionics technician and Lowry AFB, CO and Homestead AFB, FL. After technical training, he was stationed in the 23rd Fighter Squadron at Spangdahlem Air Base, Germany. While stationed in Europe, he earned a Bachelor of Science degree in Computer and Information Science from the University of Maryland. He was selected to attend the Air Force Officer Training School (OTS) in May 1995, and received his commission in August of that year.

Following graduation from OTS, Captain Breighner attended the Basic Communications Officer Training course at Keesler AFB, MS. Since his commissioning, Captain Breighner has served in assignments at Offutt AFB, NE and Whiteman AFB, MO. In August 1999, Captain Breighner entered the Air Force Institute of Technology as a graduate student in the computer science program.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 20-03-2001		2. REPORT TYPE MASTER'S THESIS		3. DATES COVERED (From - To) JUN 00 - MAR 01	
4. TITLE AND SUBTITLE A SEMANTIC INTERFACE TO SCENARIO COMPONENT REUSE IN DOD SIMULATION SYSTEMS				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
				5d. PROJECT NUMBER	
6. AUTHOR(S) LAWRENCE A. BREIGHNER, CAPT, USAF				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P. Street, Bldg 640 Wright-Patterson AFB, OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/01M-01	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Sensors Directorate Attn: Foster, R.M. Bldg 620 S1D34 2241 Avionics Circle Wright-Patterson AFB, OH 45433-7303 DSN: 785-2811 x4364				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/SNZW	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release, Distribution Unlimited					
13. SUPPLEMENTARY NOTES Advisor: Major Michael L. Talbert, DSN 785-6565 x4280, michael.talbert@afit.edu					
14. ABSTRACT The Department of Defense utilizes various simulation systems to model employment of forces and weapons systems in operational environments. The data files that model these environments and weapons systems are extremely large and complex, and require many person-hours to develop. Compounding the problem, these data files are distributed across multiple systems in a heterogeneous environment. Currently, there is no automated means of identifying and retrieving reusable portions of these files for reuse in a new scenario under development. This work develops a multi-agent system that catalogs the files, and provides the user with a means of identifying and retrieving reusable components. Additionally, since the format of the source files varies from simulator to simulator, a process for performing scenario component transformation is developed and implemented.					
15. SUBJECT TERMS Heterogeneous databases, Simulations, Agents, Semantic Modeling, Object-Oriented, Agent-Oriented Information Systems, SUPPRESSOR, SWEG, Reuse, Signature Analysis, Information Retrieval, Transformation					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 166	19a. NAME OF RESPONSIBLE PERSON Major Michael L. Talbert
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (include area code) DSN 785-6565 x4280